

# **Generic Adaptation Support for Wireless Sensor Networks**

Dissertation

zur Erlangung des akademischen Grades eines  
Doktors der Naturwissenschaften  
(Dr. rer. nat.)

durch die Fakultät für Wirtschaftswissenschaften der

Universität Duisburg-Essen  
Campus Essen

vorgelegt von: Daniel Minder

geboren in: Aalen

Essen 2015

Tag der mündlichen Prüfung: 12.04.2016

Erstgutachter: Prof. Dr. Pedro José Marrón

Zweitgutachter: Prof. Dr. Jörg Hähner

# Abstract

Wireless Sensor Networks are used in various and expanding application scenarios and are also considered to be important elements of the Internet of Things. They monitor and deliver data, which is not only used for research but to an increasing degree also in business environments. With the increasing complexity of these scenarios and the increasing dependency on the availability of the sensor network data, the requirements to a Wireless Sensor Network increase at the same pace.

Since Wireless Sensor Networks are typically implemented using resource-constrained platforms, sensor network algorithms are typically optimised for specific operating conditions such as static or mobile networks, high or low traffic etc. However, due to scenario complexity and dynamic real-world conditions a static configuration of a Wireless Sensor Network software cannot always meet the requirements. Moreover, these requirements of the sensor network's user can change over time, for example concerning accuracy. Therefore, the sensor network software has to adapt itself to cope with dynamic system conditions and user requirements.

This thesis presents the TinyAdapt and TinySwitch frameworks to solve the aforementioned problems. TinyAdapt, our generic adaptation framework for Wireless Sensor Networks, allows for the autonomous adaptation of arbitrary sensor network algorithms based on explicit and intuitively defined user preferences and on automatically monitored network conditions. Due to a two-phase approach, run-time adaptation is executed completely and efficiently on standard sensor node hardware and does not need support from, e.g., the base station. The creation of adaptive applications is guided by a complete workflow, which is presented as well.

When changing parameters of an algorithm is not enough to achieve the desired adaptation results, the algorithm has to be exchanged completely. However, several limitations of TinyOS and the sensor node hardware limit the use of simple code exchange by node reprogramming for efficient adaptation. TinySwitch, our generic switching framework, allows to switch between alternative algorithms that are already installed in parallel. TinySwitch analyses these algorithms, determines their dependencies and creates all code to enable one of the algorithms while isolating all others. Due to its minimal overhead, TinySwitch is perfectly suited for run-time adaptation in TinyAdapt.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contribution . . . . .	3
1.2	Structure . . . . .	4
1.3	Publications . . . . .	5
<b>2</b>	<b>Background and Related Work</b>	<b>9</b>
2.1	Wireless Sensor Networks . . . . .	9
2.1.1	Sensor Nodes . . . . .	9
2.1.2	Hardware Platforms for Research . . . . .	10
2.1.3	Wireless Sensor Network Characteristics . . . . .	11
2.1.4	Operating Systems . . . . .	14
2.2	TinyCubus . . . . .	19
2.2.1	Tiny Cross-Layer Framework . . . . .	20
2.2.2	Tiny Configuration Engine . . . . .	21
2.2.3	Tiny Data Management Framework . . . . .	22
2.3	Related Work . . . . .	22
2.3.1	Application (Re-)Configuration . . . . .	22
2.3.2	Code Exchange . . . . .	26
2.3.3	Monitoring Systems . . . . .	28
2.3.4	Mobility Metrics . . . . .	30
2.4	Conclusion . . . . .	32
<b>3</b>	<b>The Generic Adaptation Framework TinyAdapt</b>	<b>33</b>
3.1	Adaptation . . . . .	33
3.1.1	Characteristics of Adaptation . . . . .	34
3.1.2	Adaptation in Computer Systems . . . . .	34
3.1.3	Adaptation in Wireless Sensor Networks . . . . .	35
3.1.4	Adaptation Possibilities . . . . .	36
3.2	Design Principles . . . . .	37
3.3	Definitions . . . . .	38
3.4	Workflow . . . . .	39
3.4.1	Algorithm Preparation . . . . .	40
3.4.2	Framework Configuration and Algorithm Exploration . . . . .	42
3.4.3	Run-time Adaptation . . . . .	44
3.5	Architecture and Data Flow . . . . .	45
3.5.1	Pre-Installation Phase . . . . .	47

3.5.2	Run-Time Phase . . . . .	48
3.6	Conclusion . . . . .	48
<b>4</b>	<b>TinyAdapt Implementation</b>	<b>51</b>
4.1	Run-Time Modules . . . . .	51
4.1.1	Distribution Component . . . . .	52
4.1.2	Settings Component . . . . .	52
4.1.3	Other Components . . . . .	54
4.2	Framework Configuration Application . . . . .	54
4.3	Monitoring . . . . .	55
4.3.1	Computation of Network Wide Maximum . . . . .	56
4.3.2	Piggybacking . . . . .	61
4.4	Mobility Metric . . . . .	63
4.5	Algorithm Exploration . . . . .	66
4.6	Configuration Table Post-Processing . . . . .	69
4.6.1	Removal of Inferior Entries . . . . .	69
4.6.2	Merging of Exploration Results . . . . .	72
4.6.3	Mapping of Network Metric Values . . . . .	73
4.7	Adaptation Engine . . . . .	73
4.8	Conclusion . . . . .	76
<b>5</b>	<b>TinySwitch - Switchable Components for TinyOS</b>	<b>79</b>
5.1	Generic Switching Concept . . . . .	79
5.1.1	Switching Architecture . . . . .	80
5.1.2	Algorithm Requirements . . . . .	81
5.1.3	Defining Common APIs . . . . .	82
5.1.4	Manual Optimisations . . . . .	83
5.2	TinySwitch Operation . . . . .	85
5.2.1	API Unification . . . . .	86
5.2.2	Wiring Analysis . . . . .	92
5.2.3	Code Generation . . . . .	100
5.3	Conclusion . . . . .	109
<b>6</b>	<b>Evaluation</b>	<b>111</b>
6.1	Experimental Setup . . . . .	111
6.2	Mobility Metric . . . . .	112
6.3	Monitoring System . . . . .	115
6.4	Case Study: Parameterisation based on User Preferences . . . . .	118
6.4.1	Algorithm Modification and Application . . . . .	118
6.4.2	Algorithm Exploration . . . . .	119
6.4.3	Run-time Results . . . . .	122
6.5	Case Study: Switching of MAC Algorithms . . . . .	124
6.6	Case Study: Switching of Routing Algorithms . . . . .	127
6.6.1	Adaptation due to Mobility . . . . .	128
6.6.2	Adaptation due to Interference . . . . .	131

6.7	Overhead Analysis . . . . .	134
6.7.1	TinySwitch . . . . .	134
6.7.2	TinyAdapt . . . . .	136
6.7.3	Comparison to Node Reprogramming . . . . .	137
6.8	Conclusion . . . . .	140
<b>7</b>	<b>Summary and Outlook</b>	<b>141</b>
7.1	Future Work . . . . .	142
	<b>Bibliography</b>	<b>145</b>





# List of Figures

2.1	Sensor node building blocks . . . . .	9
2.2	BlinkC.nc module . . . . .	16
2.3	BlinkAppC.nc configuration . . . . .	16
2.4	Wiring of the Blink application . . . . .	17
2.5	Contiki Blink application . . . . .	18
2.6	Proposed TinyCubus architecture . . . . .	19
3.1	The TinyAdapt workflow and involved actors . . . . .	40
3.2	The TinyAdapt framework architecture and data flow . . . . .	46
4.1	Computation of network-wide maximum . . . . .	57
4.2	Calculation of network-wide maximum example: basic operation . . . .	58
4.3	Calculation of network-wide maximum example: update of maximum .	59
4.4	Calculation of network-wide maximum example: timeout . . . . .	60
4.5	TinyOS network stack with Piggybacking . . . . .	62
4.6	Format of a Piggyback packet . . . . .	63
4.7	Irreducible Configuration Table entries . . . . .	71
4.8	Reduction of Configuration Table . . . . .	71
4.9	Adaptation Algorithm . . . . .	74
5.1	General switching architecture . . . . .	80
5.2	Different calls to and from algorithm . . . . .	81
5.3	Unification of interfaces for CTP and Flooding . . . . .	90
5.4	Algorithm, components and connections . . . . .	93
5.5	Supported connections between external and algorithm components . .	94
5.6	Algorithm component D depends transitively on external component A	95
5.7	Pseudocode of command to initiate switching . . . . .	103
5.8	State machine of the multiplexing layer tracking algorithm state and implementing switching . . . . .	105
5.9	Multiplexing and isolation layers for routing algorithms CTP and Flooding	110
6.1	Calculated ranges of the metric values for different $\alpha$ . . . . .	113
6.2	Real value range of the final metric with $\alpha = 0.3$ . . . . .	114
6.3	Value range of the metric's global maximum . . . . .	114
6.4	Convergence time and the fail ratio with various broadcast intervals . .	115
6.5	Convergence time and average flooding time for different velocities with Random Waypoint (left) and Manhattan (right) Model . . . . .	116

## List of Figures

6.6	Average number of failing nodes (left) and fail ratio (right) for different velocities with Random Waypoint and Manhattan Mobility Models . . .	117
6.7	Maximum deviation from network-wide maximum over time . . . . .	117
6.8	Influence of data packet and LPL interval on the delivery ratio for Security Monitoring Application . . . . .	120
6.9	Influence of data packet and LPL interval on the energy for Security Monitoring Application . . . . .	121
6.10	Influence of data packet and LPL interval on the latency for Security Monitoring Application . . . . .	121
6.11	Delivery ratio over time for adaptive security monitoring application . .	122
6.12	Overall network energy over time for adaptive security monitoring application . . . . .	123
6.13	Latency over time for adaptive security monitoring application . . . . .	124
6.14	Number of received packets in horse monitoring scenario for applications performing no adaptation, parameter-based adaptation and using TinySwitch . . . . .	126
6.15	Network energy consumption in horse monitoring scenario for applications performing no adaptation, parameter-based adaptation and using TinySwitch . . . . .	127
6.16	Delivery ratio in the dynamic scenario with TinyAdapt . . . . .	130
6.17	Energy requirement in the dynamic scenario with TinyAdapt . . . . .	131
6.18	Delivery ratios for pure CTP/Flooding applications with interferers between time A and B . . . . .	132
6.19	Delivery ratios for adaptive application using TinyAdapt and TinySwitch with interferers between time A and B and adaptation at C and D . . .	133
6.20	Total number of packets sent by different routing applications . . . . .	133
6.21	Setup for power measurement of sensor nodes . . . . .	138
6.22	Current drain of MicaZ node during reprogramming . . . . .	138
6.23	Current drain of TelosB node during reprogramming . . . . .	139

# List of Tables

2.1	Comparison of application (re-)configuration systems . . . . .	25
2.2	Comparison of code exchange systems . . . . .	29
2.3	Comparison of monitoring systems . . . . .	30
2.4	Comparison of mobility metrics . . . . .	31
3.1	Comparison of adaptation methods . . . . .	37
3.2	Example Goal Definition . . . . .	45
4.1	Adaptive exploration of example application — static scenario . . . . .	68
4.2	Brute-force exploration of example application — mobile scenario . . . . .	68
4.3	Characteristics Table of example application after reduction . . . . .	72
4.4	Example Goal Definition . . . . .	75
4.5	Adaptation process for static example application . . . . .	75
4.6	Application of the relaxation rules in example . . . . .	76
4.7	Adaptation process for mobile example application . . . . .	76
5.1	Overview of generated code for switching of routing algorithms . . . . .	109
6.1	Algorithm parameters for security monitoring application . . . . .	119
6.2	Overview generated code for MAC algorithm switch . . . . .	125
6.3	Goal Definition for Logistics Application . . . . .	129
6.4	Comparison of power consumption without (N-S, N-M) and with TinyAdapt (A-S, A-M) in different scenarios . . . . .	129
6.5	Analysis of multiplexing and isolation layer for case studies 2 and 3 . . . . .	135
6.6	TinyAdapt (module) memory usage and complete size for Case Study 3 . . . . .	136



# 1 Introduction

Having started in the 1980's as a military project at the Defense Advanced Research Projects Agency (DARPA) in the U.S., Wireless Sensor Networks (WSNs) entered civil use at the beginning of the twenty-first century. Monitoring the nests of birds (storm petrels) on an U.S. island in 2002 [MCP<sup>+</sup>02] is considered to be the first real-world WSN deployment. Since then, WSNs spread to more and more areas: several other animal monitoring applications have been developed, for example for zebras [LM03] or for wild horses in the PLANET project [PLA], as well as applications for environmental monitoring (e.g. micro-climate measurements on redwood trees [TPS<sup>+</sup>05]), agriculture (e.g. vineyard monitoring [BBB04]), structural health monitoring (e.g. the monitoring of a medieval tower in Trento [CMP<sup>+</sup>09]), industry (e.g. monitoring underground pipes [SNMT07]), logistics (e.g. cold chain management [RV04]), health care (both in clinical environment, e.g. [KLC<sup>+</sup>10], and as body sensor networks, e.g. fall detection [AYO<sup>+</sup>10]), civilian surveillance (e.g. fence monitoring [WTV<sup>+</sup>07]), smart cities (e.g. real-time parking availability in San Francisco [San14]) and smart buildings (both as monitoring devices only for, for example, smart metering [JVLT<sup>+</sup>09] and as sensor-actuator-network, for example, for HVAC [DGM05]).

Currently, most of the given examples are also considered to be Internet of Things applications with the sensors delivering data that is stored “in the cloud” and then analysed further and acted upon. However, it is also acknowledged that due to energy considerations the individual sensors might not be able to directly connect to mobile phone infrastructure such as LTE or LTE-A to deliver their data [LAAZ14]. A possible solution is to connect locally organised sensor networks to the internet via gateways. Therefore, WSNs will also stay relevant for the coming decades as “edge” drivers for the Internet of Things [VFG<sup>+</sup>14].

Typical sensor networks consist of a multitude of single sensor nodes. In the vision of “Smart Dust” [KKP99] there will even be hundreds or thousands of them, which seems feasible soon due to progress in miniaturisation and nanotechnologies [MCM15]. Thus, a single sensor node has to be very cheap. Moreover, in most of these applications, sensor nodes operate autonomously powered by batteries, limiting available energy. Due to these cost and energy constraints, small and efficient microcontrollers are typically used on common sensor node platforms, which further limits available resources such as processing power and available memory.

## 1 Introduction

To cope with these resource constraints, many algorithms that have been developed for traditional distributed systems need to be redesigned. For example, the Network Time Protocol for time synchronisation requires the exchange of several big messages, which would consume too much energy and memory on sensor nodes. In contrast, Reference Broadcast Synchronisation (RBS) [EGE02] exploits the broadcast nature of sensor network's wireless channel and synchronises a set of receiver with one another, thus removing the send and access time of the sender in the time offset calculation.

In these redesigned algorithms, trade-offs often have to be made with respect to antagonistic requirements, for example power consumption vs. latency or accuracy. In the time synchronisation example, either the synchronisation accuracy can be increased by sending more synchronisation packets, or the power consumption can be decreased by reducing the amount of packets sent, without changing the rest of the algorithm. Moreover, most redesigns are based on certain assumptions on the network model to achieve further resource savings, e.g. sparse or dense deployments, static or mobile scenarios. For example, tree-based routing algorithms assume stable links that usually occur in static scenarios only.

WSN applications consist of many such algorithms, each of which carries out a specific functionality and which compete for limited energy, memory, processing and bandwidth. For example, a remote sensing application can include a data collection algorithm, a data processing algorithm, a routing algorithm, etc. Due to the variety of algorithms and the multitude of their configuration parameters choosing a suitable set of algorithms and parameters can be a tedious and difficult task to do manually. Also, all the optimisations described previously must take place for an application as a whole since changes in only some parts might affect the overall behaviour. Moreover, the interrelation between high-level end-user preferences (e.g. regarding power consumption or accuracy) and low-level algorithm parameters (e.g. maximum packets sent or transmission power) might not be obvious.

Management of WSNs often does not end with their installation since many of them should operate for a longer time without human intervention on a technical level. However, the requirements of the user towards the application and/or the network conditions could change during the lifetime of a sensor network, both instantaneously or gradually. To achieve optimal performance again the WSN would need reconfiguration. For example, a logistics application could track the position of goods at a logistics centre, where nodes are highly mobile. However, when the goods are packed in a container they can form a static network with a single gateway. During operation, the expectations of the customer could rise, requiring to deliver information in near real-time. For both changes, reconfiguration would be necessary for optimal performance.

To solve the aforementioned problems, this thesis proposes a generic framework and a complete workflow to facilitate and drive the creation and management of adaptive WSN applications. It eases the selection of algorithms and their parameters for the initial configuration of WSN applications and enables efficient run-time adaptation

of WSN applications based on both user preferences and network conditions. Run-time adaptation can be achieved by both changing only parameters and complete algorithms. For the exchange of algorithms, this thesis also proposes a novel framework that allows to integrate multiple alternative algorithms in one TinyOS application and to efficiently switch between these alternatives during run-time.

## 1.1 Contribution

This thesis has two main contributions: Firstly, it presents a generic adaptation framework for Wireless Sensor Networks, TinyAdapt, which allows for the autonomous adaptation based on user preferences and network conditions. Secondly, a switching framework for TinyOS is presented that allows to switch between alternative algorithms during run-time.

Existing approaches for the reconfiguration of sensor networks consider it as a service composition problem only [KNE<sup>+</sup>04, MPS08, AMM<sup>+</sup>12] and/or require a powerful base station for complex optimisation calculations [MRAM09]. Distributed approaches are based on (partly hard-coded) scripts that have to be created manually without support from the development environment [FET<sup>+</sup>10, SGC13].

Starting with general reflections on adaptation, we examine adaptation strategies for WSNs. Then, the architecture of the generic adaptation framework TinyAdapt is developed. Based on three main design principles, TinyAdapt features autonomous adaptation through parameterisation and code exchange based on network dynamics and explicit and intuitive user preferences. Due to a two-phase approach where algorithms and parameters are evaluated before installing the final application, the run-time components run completely on standard sensor node hardware and perform efficient adaptation without support from a base station or other resource-rich devices. What is more, TinyAdapt does not limit the type of algorithms that it can handle. This work has been published in [MHM10].

The whole process of creating an adaptive application is steered by an overall workflow, which is also described in this thesis. Several steps of this workflow are facilitated by the TinyAdapt framework, such as the configuration of the network monitoring, the evaluation of the algorithms and parameters, and the post-processing of all evaluation results. Since mobility is an important network characteristic and is often affecting algorithms, we also develop a new mobility metric that is less dependent on the total number of nodes. Also, a prototypical implementation of TinyAdapt is shown for TinyOS.

Performance enhancement achieved by parameter changes is usually limited since they do not change the general logic of an algorithm. Instead, a completely different algorithm would be a better option, which requires partial or complete reprogramming of the WSN in TinyOS. This has been achieved by different solutions such as [HC04, GMN09].

## 1 Introduction

Since reprogramming is slow, energy consuming and can lead to node failures, this thesis proposes TinySwitch where all alternative implementations of a task are included in one single binary image, while ensuring that exactly one selected instance for each task is active at a time and all others remain inactive. Although switching solutions where proposed earlier [KNE<sup>+</sup>04, AMM<sup>+</sup>12, SGC13] they are either limited to certain algorithm types, do not assist the developer in finding the boundaries and interfaces of an algorithm and/or leave the implementation of the switch to the developer.

Our solution, the TinySwitch framework, supports the developer in creating such an application by analysing the interfaces of the alternative implementations and by generating code to implement aforementioned switching features. It ensures that all non-active algorithms do not interfere with the rest of the application. Moreover, TinySwitch is lightweight, adding only minimal run-time overhead.

## 1.2 Structure

The rest of the thesis is structured as follows: Chapter 2 introduces Wireless Sensor Network hardware and operating systems, which are relevant for this thesis. Then, it analyses adaptation in general, for computer systems and for sensor networks in general and derives methods for adaptation. The chapter also includes an overview of the related work in adaptation, code exchange, and mobility metrics.

The generic adaptation framework is presented in Chapter 3. First, the design principles of TinyAdapt are established. Based on these principles, the overall workflow to create an adaptive application and the architecture and data flow of TinyAdapt is developed.

Chapter 4 shows an implementation of TinyAdapt for TinyOS by presenting concrete software modules for each part of the architecture and workflow. Here, the monitoring of network characteristics, the mobility metric, the post-processing of the evaluation results and the actual adaptation process are detailed.

The algorithm switching framework TinySwitch is presented in Chapter 5, which includes the analysis of the connections of an algorithm, the automatic creation of code for a multiplexing and isolation layer, and a state machine to handle the complex start/stop of algorithms by the multiplexer in event-based systems.

In Chapter 6, TinyAdapt and its sub-parts Mobility Metric and Monitoring System and TinySwitch are evaluated. To do so, three different case studies are presented that show parameter based adaptation, the direct switching of MAC algorithms by the application, and the autonomous adaptation by switching routing algorithms to cope with mobility or interference.

The thesis concludes with Chapter 7 that summarises the contributions and gives an outlook on possible extensions.



## 1.3 Publications

In the following, all publications in which the author of this thesis was involved are listed. This includes publications on TinyCubus and its single parts, on TinyAdapt, and other publications.

### TinyCubus with FlexCup and TinyXXL

- Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Kurt Rothermel and Christian Becker. Adaptation and Cross-Layer Issues in Sensor Networks. In *Proceedings of the First International Conference on Intelligent Sensors, Sensor Networks & Information Processing (ISSNIP 2004)*, page 623-628, December 2004.
- Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter and Kurt Rothermel. TinyCubus: A Flexible and Adaptive Framework for Sensor Networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, page 278-289, January 2005.
- Pedro José Marrón, Daniel Minder, Andreas Lachenmann and Kurt Rothermel. TinyCubus: A Flexible and Adaptive Cross-Layer Framework for Sensor Networks. In *4th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, Technical Report TR 481, Computer Science Department, ETH Zurich, page 49-54, March 2005.
- Pedro José Marrón, Daniel Minder, Andreas Lachenmann, Olga Saukh and Kurt Rothermel. Generic Model and Architecture for Cooperating Objects in Sensor Network Environments. In *Proceedings of the 12th International Conference on Telecommunications (ICT 2005)*, May 2005.
- Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh and Kurt Rothermel. Adaptive System Software Support for Cooperating Objects. In *Proceedings of the Workshop on Smart Object Systems, September 2005. In conjunction with the 7th International Conference on Ubiquitous Computing (UbiComp 2005)*.
- Andreas Lachenmann, Pedro José Marrón, Daniel Minder and Kurt Rothermel. An Analysis of Cross-Layer Interactions in Sensor Network Applications. In *Proceedings of the Second International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP '05)*, page 121-126, December 2005.
- Pedro José Marrón, Daniel Minder, Andreas Lachenmann and Kurt Rothermel. TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks. it - Information Technology , vol. 47 no. 2, page 87-97, 2005.

## 1 Introduction

- Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Matthias Gauger, Olga Saukh and Kurt Rothermel. Management and configuration issues for sensor networks. *International Journal of Network Management*, Special Issue: Wireless Sensor Networks , vol. 15 no. 4, page 235-253, 2005.
- Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh and Kurt Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN '06)*, page 212-227, February 2006.
- Andreas Lachenmann, Pedro José Marrón, Daniel Minder, Matthias Gauger, Olga Saukh and Kurt Rothermel. TinyXXL: Language and Runtime Support for Cross-Layer Interactions. In *Proceedings of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '06)*, page 178-187, September 2006.
- Pedro José Marrón, Daniel Minder, Andreas Lachenmann, Olga Saukh and Kurt Rothermel. Generic Model and Architecture for Cooperating Objects in Sensor Network Environments. *African Journal of Information & Communication Technology* , vol. 2 no. 1, page 1-11, 2006.

### TinyAdapt

- Daniel Minder, Pedro José Marrón, Andreas Lachenmann and Kurt Rothermel. Coordinated group adaptation in sensor networks. In *6th Fachgespräch Sensornetze*, Technical Report AIB 2007-11, RWTH Aachen, page 43-46, July 2007.
- Daniel Minder, Andreas Grau and Pedro José Marrón. On group formation for self-adaptation in pervasive systems. In *Proceedings of the First international conference on Autonomic computing and communication systems (Autonomics '07)*, page 1-10, October 2007.
- Daniel Minder, Marcus Handte and Pedro José Marrón. TinyAdapt: An Adaptation Framework for Sensor Networks. In *Proceedings of the 7th International Conference on Networked Sensing Systems (INSS 2010)*, page 253-256, June 2010.

### Other publications

- Daniel Minder, Pedro José Marrón, Andreas Lachenmann and Kurt Rothermel. Experimental construction of a meeting model for smart office environments. In *Proceedings of the First Workshop on Real-World Wireless Sensor Networks (REALWSN 2005)*, SICS Technical Report T2005:09, June 2005.
- Olga Saukh, Pedro José Marrón, Andreas Lachenmann, Matthias Gauger, Daniel Minder and Kurt Rothermel. Generic Routing Metric and Policies for WSNs. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN '06)*, page 99-114, February 2006.

- Pedro José Marrón, Daniel Minder and Embedded WiSeNts Consortium, editor. Embedded WiSeNts Research Roadmap. Logos, Berlin 2006.
- Andreas Lachenmann, Pedro José Marrón, Daniel Minder, Olga Saukh, Matthias Gauger and Kurt Rothermel. Versatile Support for Efficient Neighborhood Data Sharing. In *Proceedings of the Fourth European Conference on Wireless Sensor Networks (EWSN 2007)*, page 1-16, January 2007.
- Andreas Lachenmann, Pedro José Marrón, Matthias Gauger, Daniel Minder, Olga Saukh and Kurt Rothermel. Removing the Memory Limitations of Sensor Networks with Flash-Based Virtual Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys 2007)*, page 131-144, March 2007. Also published in ACM SIGOPS Operating Systems Review, vol. 41(3), 2007.
- Andreas Lachenmann, Pedro José Marrón, Daniel Minder and Kurt Rothermel. Meeting Lifetime Goals with Energy Levels. In *Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys '07)*, page 131-144, November 2007.
- Matthias Gauger, Pedro José Marrón, Marcus Handte, Olga Saukh, Daniel Minder, Andreas Lachenmann and Kurt Rothermel. Integrating Sensor Networks in Pervasive Computing Environments Using Symbolic Coordinates. In *Proceedings of the Third International Conference on Communication System Software and Middleware (COMSWARE 2008)*, January 2008.
- Matthias Gauger, Daniel Minder, Pedro José Marrón, Arno Wacker and Andreas Lachenmann. Prototyping Sensor-Actuator Networks for Home Automation. In *Proceedings of the 3rd Workshop on Real-World Wireless Sensor Networks (REALWSN 2008)*, April 2008.
- Pedro José Marrón, Stamatis Karnouskos, Daniel Minder and the CONET Consortium, editor. Research Roadmap on Cooperating Objects. Office for Official Publications of the European Communities, Luxembourg 2009.
- Pedro José Marrón, Stamatis Karnouskos, Daniel Minder and Aníbal Ollero, editor. The Emerging Domain of Cooperating Objects. Springer, Berlin, Heidelberg 2011.
- Pedro José Marrón, Daniel Minder and Stamatis Karnouskos. The Emerging Domain of Cooperating Objects - Definition and Concepts. Springer, Berlin, Heidelberg 2012.
- Stamatis Karnouskos, Pedro José Marrón and Daniel Minder. Cooperating Objects Design Space and Markets. In *4th International Workshop on Networks of Cooperating Objects for Smart Cities 2013 (CONET/UBICITEC 2013)*, April 2013.



# 2 Background and Related Work

This chapter contains background information relevant for the concepts and systems developed in this thesis. It first describes hardware and software for Wireless Sensor Networks. Then, the TinyCubus project is presented, in which TinyAdapt and TinySwitch have been developed. Finally, related work for adaptation, code exchange and mobility metrics is discussed.

## 2.1 Wireless Sensor Networks

Wireless Sensor Networks (WSN) are formed by several sensor nodes that gather, process and transmit data from the physical world in a distributed, cooperative and unobtrusive way. We show the typical design of a sensor network, the concrete platforms used in this thesis, the general characteristics of WSN software and algorithms, and give an introduction into WSN operating systems.

### 2.1.1 Sensor Nodes

Typical sensor nodes consist of a microcontroller, a communication device, sensors and possibly actuators, and a power supply (see Figure 2.1). Often, especially on experimental platforms used in research, flash memory to persistently store data and connectors to program the sensor nodes from the PC are present (not shown in the figure).

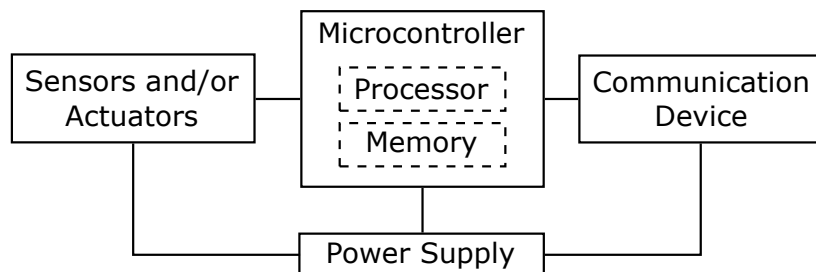


Figure 2.1: Sensor node building blocks

## 2 Background and Related Work

The microcontroller contains, among other things, a processor and memory, usually both flash ROM to store the program code and normal RAM. It executes the stored program and interacts with the communication device, sensors and actuators to control them and to process data. Several peripherals are usually present on a microcontroller as well, for example a timer, digital and analogue I/O channels, and special I/O busses such as USART or I<sup>2</sup>C.

Instead of a microcontroller, a microprocessor and external memory could be used — however, this setup would consume more power and the peripherals have to be provided externally. Recently, sensor nodes based on FPGAs (field programmable gate array) have been shown [LSKT13]. If wireless sensor nodes for a specific application are to be produced in larger quantities, it is also usually cheaper to realise them as ASIC (application-specific integrated circuit). An example for this is the in-tire pressure sensing system presented in [FDH<sup>+</sup>09].

Data from the physical world is usually gathered by different sensors. The type of sensor depends on the physical phenomenon being observed. It can range from simple sensors such as temperature, humidity and the photosynthetically active radiation used by the micro-climate monitoring on redwood trees [TPS<sup>+</sup>05] to highly specialised and custom-made sensors such as fibre optic deformation sensors used in the Torre Aquila deployment [CMP<sup>+</sup>09]. Some simple sensors can be evaluated directly by the microcontroller using its internal Analogue-Digital-Converter (ADC) while other sensors are connected via different buses such as I<sup>2</sup>C or SPI. Some experimental platforms, e.g. the TelosB platform shown in the next section, are already equipped with simple sensors.

Actuators are rarely used since they would consume too much power. On experimental platforms simple actuators like LEDs or buzzers can be found. For more sophisticated actuation, the sensor nodes are often connected to resource-rich platforms such as robots.

Data is sent and received by a communication device, typically a radio transceiver. Current experimental platforms mostly use the IEEE 802.15.4 standard, but also communication via Bluetooth or proprietary protocols in lower ISM bands can be found. System-on-Chip solutions integrate microcontroller and radio chip, such as the CC2530 chip from Texas Instruments.

All of these devices are powered by a central power source, which is usually a battery. Energy harvesting can only be used in certain scenarios (e.g. the in-tire pressure monitoring [FDH<sup>+</sup>09]).

### 2.1.2 Hardware Platforms for Research

Two wireless sensor node platforms have been available early for research, and still they are prominent representatives for a class of small and resource-constrained devices:

MicaZ and TelosB. For this reason, this thesis focuses on their characteristics and also uses them for evaluation.

The MicaZ node is based on the Mica2 node [HHKK04] and differs only in the radio module. At the core of both nodes, the ATmega128L, an 8-bit RISC microcontroller from Atmel, is running at  $\approx 7.37$  MHz. The microcontroller follows the Harvard architecture with 128kB of program flash memory, 4kB of RAM and additional 4kB of EEPROM for easy storage of small configuration data. 8 ADC channels and different serial channels can be used to connect external sensors directly to the microcontroller. An external 512kB flash memory is connected where larger amounts of data, e.g. sensor readings or received data, can be stored.

The predecessor, Mica2, used proprietary communication in the ISM or SRD band around 315, 433, 868 or 915 MHz with a MAC layer completely implemented in software. MicaZ nodes are now based on 802.15.4 communication, which is completely realised in an extra radio chip (CC2420 from ChipCon/Texas Instruments) and, thus, reduces processor load. MicaZ achieves a radio data rate of 250kbps compared to 19.2kbps for Mica2.

The second node, TelosB [PSC05], is based on Texas Instrument's MSP430F1611, which is a 16-bit RISC microcontroller running at 4 MHz. It contains 48kB of flash memory, 10kB of RAM and 256 bytes of extra flash memory to store configuration data. Although the MSP430 follows the von-Neumann architecture, the executable programs are typically stored completely in the flash memory and executed from there since RAM contents will be lost when the controller is reset. The external flash has a size of 1MB, but is internally organised in a different way compared to the Mica2/MicaZ flash chip.

The TelosB uses the same radio chip as the MicaZ nodes. In contrast to MicaZ, the TelosB already includes a user button, which can be queried from software, and two light and one humidity and temperature sensor. Both platforms are equipped with three LEDs that can be used for simple debugging.

Both MicaZ and TelosB are powered by two AA batteries. However, TelosB's MSP430 processor consumes less power than the ATmega128L: at 3V the supply current in active mode for the MSP430 is typically 2mA while the ATmega128L requires around 8mA.

### 2.1.3 Wireless Sensor Network Characteristics

Typically, tens to hundreds of the previously described wireless sensor nodes form a Wireless Sensor Network (WSN). In the vision of "Smart Dust" [KKP99] there will even be thousands of these nodes. The special features of WSNs emerge when the nodes start to network and cooperate with each other. Depending on the concrete application, the WSN gathers sensor data, processes, distributes and stores this data in the network, and/or delivers the data to one or more distinct nodes (called "sinks")

## 2 Background and Related Work

or “base stations”) from where it is forwarded to a connected PC or other non-WSN platform.

Due to the small form factors of the single nodes and their autonomous and self-sustaining nature a WSN can achieve a much more dense observation of physical phenomena than traditional, possibly tethered methods. Moreover, measured data can be verified using data from neighbouring nodes and, thus, fusion of many data points can improve data quality. The redundancy of dense deployments can also lead to higher resilience to node failures since other nodes can take over. It can be noted that the single node does not matter but the data and usually the location it comes from is more important.

However, the same characteristics pose fundamental challenges to the development of WSN software. First, energy is limited due to use of batteries, and even energy-harvesting does not deliver abundant power. This energy restriction and the lower costs are the main reason for using low-power devices such as simple RISC microcontrollers, which leads to further resource restrictions concerning processing speed and memory. Finally, radio communication consumes a lot of power, which needs to be considered under the aspect of limited energy.

For these reasons, software for WSN needs to be optimised for resource-constrained systems and both operating systems (see next section) and algorithms need to be (re-)designed carefully. Some general approaches can be noted:

- Devices on the sensor nodes are switched off or put to sleep mode as often as possible. This is especially important for the radio transceiver, but also for sensors and the microcontroller. However, all algorithms have to cope with the fact that other nodes might be switched off temporarily.
- In dense deployments, redundancy can be exploited to execute tasks in an alternating way while still maintaining radio connectivity and sensing coverage.
- Data is pre-processed (mainly aggregated) in the network to reduce the amount of transmitted data as much as possible.
- Nodes need to collaborate to fulfil a task at all or to improve data quality.

Algorithm design always happens while taking into account the deployment conditions. As mentioned before, some optimisations are only possible in dense deployments, others if the network is static, i.e. the nodes do not move (or are moved). We give some prominent examples for such optimisations under specific conditions in the MAC and Routing Layer.

### MAC Layer

If nodes are sleeping at random times communication between two nodes is not always possible. A possible solution is to synchronise the nodes and to establish a schedule when each node is allowed to send. This idea is, for example, followed in LEACH



[HCB00]. Its operation is divided into rounds, which consist of a setup phase and a steady phase. During the setup phase, clusters are formed: some nodes decide to be cluster heads and all other nodes join one of these clusters (the exact process is not important at this point). Then, the cluster head creates and distributes a schedule when each of the nodes belonging to its cluster is allowed to send a packet to the cluster head during the steady phase.

This TDMA (time division multiple access) protocol avoids collisions and enables the nodes to wake up only during their scheduled time, thus saving energy. However, setting up the transmission schedules (and for LEACH also the clusters) creates overhead, which can be comparatively high if data traffic is low. Moreover, the radio channel is not used in an optimal way if a few nodes in a cluster have to send more data than all other nodes.

The standard MAC layer of TinyOS follows a different approach by implementing BoX-MAC(-2) [ML08]. Sender and receiver know the wake-up intervals of each other, but are not synchronised. When the sender wants to send a packet, it repeats this packet continuously during the wake-up interval. Once the receiver wakes up it checks the channel, detects energy on the channel and starts receiving the packet. If the packet is intended for this receiver, an acknowledgement packet is sent back. Then, the sender stops resending the packet and goes back to sleep if there are no other packets to send. Also, the receiver will go back to sleep if the channel is free or if a packet is not destined for this receiver.

Although this protocol saves a lot of energy, it does not perform optimally under high load. First, due to the contention-based nature the probability for collisions increases. And second, since the channel is rarely free, nodes cannot go to sleep early after checking the channel energy but have to receive the packet to check the destination address. Therefore, in such cases a scheduled-based protocol as presented before can be advantageous.

As can be seen from these examples, MAC protocols can exhibit different performance depending on the system conditions, here traffic and indirectly node density since channel allocation in a certain area also increases with the number of nodes in that area.

## Routing

A typical traffic pattern in WSN is the transmission of sensor readings to one or more base stations where the data is forwarded to a PC evaluating the data. The Collection Tree Protocol CTP [FGJ<sup>+</sup>07] of TinyOS supports this in an optimal way for static and stable networks. The basic idea is to create spanning trees with the base stations as the root nodes of these trees. Then, data is only sent along this tree, which also allows for data aggregation in the tree.

## 2 Background and Related Work

Each node estimates the bi-directional quality of the links to its neighbours. To do so, it calculates the ETX (estimated transmissions) metric for each link based on regular beacon broadcasts and the success rate of data transmissions. The root nodes start to distribute their ETX value of 0 in the beacon packets. Each node receiving a beacon calculates the resulting path metric as the sum of the received ETX value from the beacon and the calculated ETX value of the link to the beacon's sender. It then selects the neighbour with the lowest ETX path value to become its parent and redistributes this ETX path value in its own beacon packets. Thus, a complete spanning tree is created with high quality paths to a base station.

If the network is stable the duration of the beacon interval increases exponentially up to a certain limit and, thus, the overhead of maintaining the tree becomes very small. However, in unstable conditions, e.g. mobile nodes, the ETX values change considerably and the nodes start resending beacons at a high rate to discover new routes to the root nodes.

With high mobility, links change with such a high rate that it is hardly possible to establish a single path from distant nodes to the base stations. If the nodes know their own location and the location of the sinks, geographic routing protocols such as GeRaF [ZR03] can be used that try to forward the packets to nodes spatially closer to the sink. Since accurate positions are usually not available on sensor nodes — GPS is very costly — flooding the packets through the whole network can be the only remaining option. Here, every node simply re-broadcasts a received packet if the packet was not seen before.

### Conclusion

Resource restrictions of sensor nodes demand the development of new WSN algorithms. These algorithms are designed for specific system conditions in which they achieve best performance. If these operating conditions do not hold performance will deteriorate or the algorithm might not be usable at all.

#### 2.1.4 Operating Systems

In general, operating systems manage the resources of a computer and provide abstractions to use these resources. This includes for example processor, memory and input-/output-devices. Typical abstractions are processes to assign processor time or address spaces to assign parts of the memory.

Operating systems for WSNs have some special characteristics. First, they have to consider the resource constraints of the WSN platforms, mainly concerning processing power, memory and energy. Second, they need to support concurrency-intensive operations since data acquisition, data processing and sending/receiving this data often occurs in parallel. Third, there exists a wide variety of sensor node platforms and as

programs should be run easily on various platforms the operating system has to ensure portability.

Two operating systems dominate the WSN area: TinyOS and Contiki. Other systems such as SOS or Mantis have been proposed, but their development has stopped since 2008/09 and, thus, we do not cover them here.

## TinyOS

TinyOS [HSW<sup>+</sup>00] appeared first on the market and is probably still the most used system. It is open source and actively developed by the TinyOS alliance. TinyOS started in 1999 at UC Berkeley, and version 1.0 was released in 2002. For version 2.0, released in 2006, large parts of TinyOS were rewritten, which required also changes in TinyOS applications. The latest version is 2.1.2 from August 2012.

TinyOS itself and its applications are written in nesC [GLv<sup>+</sup>03], an extension of C, which was designed especially to support the TinyOS abstractions such as modules, configurations, interfaces, commands and events. In TinyOS, software is arranged in components, which can be either modules or configurations: modules contain the actual implementation and configurations describe the connections between modules or other configurations. While inside a module normal functions can be used, only commands and events can be used from external components. Related commands and events are grouped in interfaces, e.g. setting and resetting a timer (commands) and the timer callback (event) in the `Timer` interface. In a TinyOS application, one component has to provide an interface and a second, connected component uses this interface. The using component can call commands in the providing component, and the providing component can signal events to the using component.<sup>1</sup>

During run-time, all events are executed atomically. If an event wants to perform a longer computation this should be moved to a so-called task. Tasks can be interrupted by events but not by other tasks, which are executed in FIFO order by the TinyOS scheduler.

Figure 2.2 shows the nesC code of a simple Blink application. It uses three interfaces. The `booted` event of the `Boot` interface is signalled after the OS initialisation. The module calls the `startPeriodic` command of the `Timer` interface to set a periodic timer of 1000ms. Then, the `fired` event is signalled regularly, and the module calls `led0Toggle` of the `Leds` interface to switch the LED on or off.

The configuration in Figure 2.3 completes the Blink application since it connects (“wires”) the different interfaces. Interfaces used in component A need to be connected to the same interface provided in component B with `A.If -> B.If;`. A configuration can also use or provide interfaces. In this case, the external interface is usually connected with “=” to the interface of a component specified inside the configuration.

---

<sup>1</sup>Components can also provide and use so-called “bare” commands and events, but this feature is rarely used.

## 2 Background and Related Work

```
module BlinkC {
  uses interface Boot;
  uses interface Timer<TMilli>;
  uses interface Leds;
}
implementation {
  event void Boot.booted() {
    call Timer.startPeriodic(1000);
  }

  event void Timer.fired() {
    call Leds.led0Toggle();
  }
}
```

Figure 2.2: BlinkC.nc module

```
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer;

  BlinkC -> MainC.Boot;
  BlinkC.Timer -> Timer;
  BlinkC.Leds -> LedsC;
}
```

Figure 2.3: BlinkAppC.nc configuration

Figure 2.4 depicts the wiring of the Blink application on the highest level. Modules are shown with single line borders, configurations with double line borders. Dashed borders indicate generic components, i.e. components that can be parameterised. The arrows indicate the connection and the direction of the arrow specifies the relation between the component providing the interface (pointed by the arrowhead) and the component using it. Note that the **BlinkAppC** configuration specifying this wiring does not appear in the figure.

When compiling a TinyOS application, only the components that are required by the application, i.e. that are connected directly or indirectly by the main configuration, are included in the final binary. Also, the compiler performs various optimisations such as inlining. Thus, in the binary the application code can hardly be distinguished from the OS code.

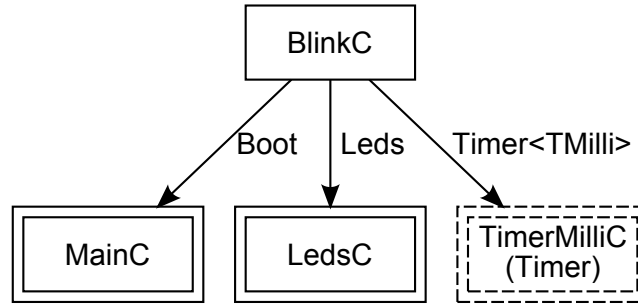


Figure 2.4: Wiring of the Blink application

## Contiki

Development of Contiki [DGV04] started in 2002 by Adam Dunkels at SICS and is now developed by a world-wide team. The latest version is 3.0 from August 2015. Contiki aims at overcoming limitations of TinyOS by supporting dynamic loading of modules and by supporting preemptive multithreading through an optional library.

During run-time, Contiki consists of the kernel, a program loader, supporting libraries and a set of processes. Processes can either be application programs or services, which provide functionality needed by more than one application process. It is possible to replace processes during run-time. In the beginning, Contiki only supported pre-linked modules that required knowledge which exact kernel is running on a sensor node. When loading such a program, only relocation is necessary. In 2006, run-time dynamic linking [DFEV06] was introduced for Contiki, which performed the linking process at run-time on the sensor node.

The kernel of Contiki is based on an event scheduler that dispatches events from a queue to the handler function of a process. Events are the only communication mechanism between processes (except for services). While synchronous events are executed immediately while the current process is waiting — thus representing kind of an inter-process procedure call — asynchronous events are put in the event queue and are executed later on in FIFO order. Event handlers run to completion and are only preempted by interrupts. Since interrupts are not allowed to post events they can set a polling flag, and the polling handler is executed after the current event has been finished.

Services implement shared functionality that can be called by other processes directly. Since the service process can also be loaded at arbitrary locations, the calling process includes a service interface stub, from where the call is forwarded to a central service layer, containing function pointers for each function. From there, the actual implementation is called. This allows for the run-time replacement of the services.

As event-driven system, Contiki also required application processes to implement the event handler as a state machine that keeps track of the current state and executes functionality and changes state based on the current state and the incoming event.

## 2 Background and Related Work

To ease programming, Protothreads [DSVA06] were introduced in 2005 that allow to write sequential programs in event-based systems. Internally, they are converted to state machines using pre-processor macros. In fact, Contiki's processes are internally also based on protothreads, offering similar functionality also for processes.

Figure 2.5 shows the Blink program in Contiki. It defines a process and starts it automatically. Inside the event handler, an infinite loop first sets the timer but then returns control to the kernel at the `PROCESS_WAIT_EVENT_UNTIL` macro. When the timer has expired, the kernel invokes the process again and execution continues by toggling the led.

```
PROCESS(blink_process, "Blink test");
AUTOSTART_PROCESSES(&blink_process);

static struct etimer et;

PROCESS_THREAD(blink_process, ev, data) {
    PROCESS_BEGIN();
    leds_off(LEDS_ALL);
    while(1) {
        etimer_set(&et, BLINK_INTERVAL);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        leds_toggle(LEDS_GREEN);
    }
    PROCESS_END();
}
```

Figure 2.5: Contiki Blink application

### Discussion

Both TinyOS and Contiki provide functionality to create standard Wireless Sensor Network applications. However, neither of them supports adaptive applications natively. While Contiki allows for the exchange of modules and starting/stopping of processes during run-time no such functionality exists in TinyOS.

Since the TinyCubus project (see next section) was started before Contiki was widely known, it is implemented using TinyOS. However, the TinyAdapt concept is general and not bound to a concrete operating system. With TinySwitch, we show a possibility for TinyOS to enable only one alternative algorithm at a time.

## 2.2 TinyCubus

The work presented in this thesis has been conducted in the TinyCubus project [MLM<sup>+</sup>05]. TinyCubus aims at creating a generic framework for TinyOS-based wireless sensor networks to cope with the complexity of real-world applications.

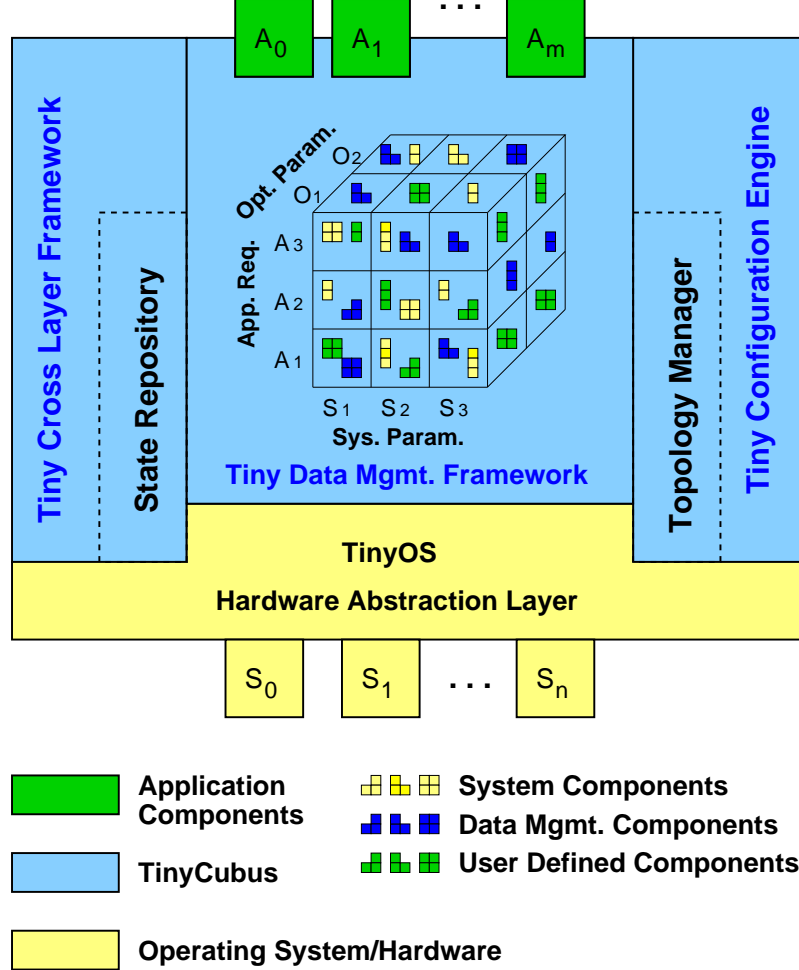


Figure 2.6: Proposed TinyCubus architecture (from [MLM<sup>+</sup>05])

The proposed architecture of TinyCubus is shown in Figure 2.6. TinyCubus is built on top of TinyOS, which acts also as hardware abstraction layer through which the sensors  $S_0, \dots, S_n$  and other hardware can be accessed. TinyCubus itself consists of three main parts: the Tiny Cross-Layer Framework, the Tiny Configuration Engine and the Tiny Data Management Framework, which are explained in the following sections. While the original proposal planned for multiple applications  $A_0, \dots, A_m$  sitting on top of TinyCubus and being managed by it, the current implementation supports one application only, which can also directly access TinyOS since algorithms need to be tailored to the characteristics of a specific applications.

### 2.2.1 Tiny Cross-Layer Framework

The Tiny Cross-Layer Framework provides a generic interface to support cross-layer interactions of components. In traditional layered approaches interaction happens only with the adjacent layers using well-defined interfaces. In cross-layer approaches, information can be exchanged between arbitrary layers, which allows for better algorithm performance, e.g. in the radio stack, and data can be shared between several layers, which reduces memory consumption, e.g. by sharing neighbourhood lists.

The implementation of the Tiny Cross-Layer Framework was shown with TinyXXL [LMM<sup>+</sup>06]. There, data is stored in a central state repository to maintain the modularity of the single algorithms and to avoid adding direct dependencies between the modules accessing the same data. TinyXXL introduces extensions to the nesC language to specify the cross-layer data and the access to it.

First, all data items that logically belong together are specified in an extra file. For example, a routing algorithm sends regular beacons for neighbour detection. Since the beacon interval should be configurable for optimisation reasons this module data is specified as follows:

```
xldata RoutingParam() {  
    int8_t BeaconInterval;  
}
```

Components reading this data declare this fact in the component header:

```
module DataRouter {  
    provides { ... }  
    uses {  
        xldata RoutingParam;  
        interface Timer;  
    }  
}  
implementation { ... }
```

Components that write the data, i.e. that are the data providers, declare data access in the **provides** section of the header. TinyXXL ensures that only one providing component exists for each data set. TinyXXL also allows to specify the costs for providing the data, and if more than one data provider exists it selects the best one according to a user specification. There is no need to create a wiring for the data like for interfaces since this is automatically done by TinyXXL.

Inside the module, the shared data can be used as normal. Using the new **ifproviding** statement code can also be only executed if a component is the data provider, thus saving resources. If immediate reaction is necessary when a parameter changes the **changed()** event for this parameter has to be implemented. For example, if the beacon



interval parameter is changed a timer could be stopped and restarted with the new rate:

```
event void RoutingParam.dataChanged() {
    call Timer.stop();
    call Timer.startPeriodic(RoutingParam.BeaconInterval * 1024);
}
```

TinyXXL also supports virtual data items that can be read like normal data but computed dynamically from other available data. This is completely transparent to the user. As extension to sharing cross-layer data on a single node TinyXXL also allows for efficient sharing of data with neighbours.

With respect to TinyCubus and in particular the Tiny Data Management Framework, which is developed in this thesis, TinyXXL is used to parameterise an algorithm or the application. The Adaptation Engine (see Section 4.7) will be the only data provider of all algorithm configuration parameters and, thus, the algorithm components will always use these parameters. This minimises code dependencies between the algorithm components and the management framework. The necessary changes to use shared cross-layer instead of private data are minimal and straight forward as can be seen from the code examples.

## 2.2.2 Tiny Configuration Engine

The main functionality of the Tiny Configuration Engine is to support self-configuration of the sensor network. For this purpose, it consists of the Topology Manager and a code distribution and installation facility. The Topology Manager allows for the assignment of roles to nodes. For example, nodes can be “source”, “aggregator” or “sink” for a simple data aggregation application. Role assignment is done based on a role specification rule set that is evaluated in a distributed fashion.

If a node requires functionality that is currently not installed, the Tiny Configuration Engine also supports the distribution and installation of code in the network. With FlexCup [MGL<sup>+</sup>06] a dynamic linking mechanism was presented that allowed for the replacement of only parts of a TinyOS binary during run-time. Since dynamic linking on the node imposes a considerable effort TinyModules [GMN09] introduced a second approach that divides the code into a static core and an exchangeable part, which has some similarity with Contiki, to reduce update size and time.

In this thesis, code update mechanisms are needed to perform adaptation through code exchange. However, since both mentioned methods still include reprogramming of the flash memory this thesis introduces algorithm switching by TinySwitch.

### 2.2.3 Tiny Data Management Framework

Finally, the Tiny Data Management Framework selects and adapts the necessary components to run the application. Different component types can be handled by the framework, such as system, data management or user defined components. These components are classified in the cube in the middle of the figure — the so-called “Cubus” — according to system parameters  $S_i$ , such as mobility; application requirements  $A_j$ , such as reliability; and optimisation parameters  $O_k$ , such as energy, communication latency or bandwidth. The core idea is to select a set of components for each required functionality that match the current set of parameters and requirements.

In this thesis, the design and implementation of the Tiny Data Management Framework is presented with TinyAdapt.

## 2.3 Related Work

In the following, approaches for application (re-)configuration, code exchange and network monitoring as well as mobility metrics are discussed. With TinyAdapt and TinySwitch, new contributions for all of these areas are presented in this thesis.

### 2.3.1 Application (Re-)Configuration

Adaptation of parameters has been applied to individual classes of WSN algorithms. For example, in the SPIN data dissemination protocol [HKB99] a node adapts by reducing its participation in the protocol if its energy approaches a low-energy threshold. The clock synchronisation of PalChaudhuri in [PSJ04] enables to tune the synchronisation accuracy based on application needs and the availability of system resources. Since these are quite tailored approaches, we are looking at more general systems now.

Applications are typically composed of different underlying services/algorithms/protocols in order to perform their operation. This configuration task can be done statically during compile time or dynamically during run-time. In the latter case, simple adaptation is possible as well.

SNACK [GKE04] is a representative of the static composition class. It provides a simplified programming language and a service library for sensor network applications to enable the easier development of efficient applications. A programmer writes an application by parameterising and combining components of the SNACK service library using a special SNACK input language, and the SNACK compiler transforms it into a regular nesC application. However, this results in a static binary where no run-time changes are possible.

Dynamic composition is achieved for example by PCOM [BHSR04]. In PCOM, applications consist of components that explicitly describe their offered and required functionality and their requirements on the platform (“contracts”). The middleware recursively creates a component tree by trying to satisfy the requirements and dependencies of components. These components can be located on multiple devices, thus forming a distributed application. Reselection of components is done by PCOM to adapt to a fluctuating set of devices or deteriorated services. PCOM is implemented in Java and, thus, needs a JVM, which is usually not available on sensor nodes.

The component abstraction and explicit specification of dependencies is also used by FiGaRo [MPS08], implemented for Contiki. A run-time system can start components after their dependencies have been satisfied, either because these are also included at startup or they are received later. FiGaRo does not provide other means to resolve the dependencies. FiGaRo also includes means to selectively distribute new components to nodes that satisfy a Boolean function of node attributes, which is the only control where and how adaptation is performed.

Adaptation goes beyond service composition (see Section 3.1). In general, adaptation systems can be classified into central and distributed systems. In central systems, the adaptation decision is taken at one device, typically the base station, which can even be an external powerful computer. In distributed systems no such distinct device is necessary, which is a more general approach.

In the central reconfiguration architecture based on GRATISplus, DESERT and GRATIS [KNE<sup>+</sup>04] a user can model several possible implementations of the same application. Each element of this application can have several properties that are combined by the system to application properties. The system selects valid configurations based on explicitly defined user constraints. During run-time, a monitoring component on each node gathers critical QoS parameters and sends them to the base station. At the base station, constraints are updated and a new design configuration is created. Reconfiguration commands are sent to the nodes specifying which components have to be stopped, rewired and started.

Also in pTunes [ZFM<sup>+</sup>12], the base station collects information about the network state, such as topology or link quality, and solves a multidimensional optimisation problem to fulfil certain application requirements concerning network lifetime, end-to-end latency, and end-to-end reliability. The new optimised MAC parameters are then distributed to the network. While the direct linkage between network state and the application requirements through mathematical models and the online solution of the optimisation problems might result in better adaptation solutions, TinyAdapt provides a more general solution suitable for arbitrary algorithms and application requirements and without the need for a central coordinator.

The system presented by Meshkova in [MRAM09] automates network protocol stack design and its run-time optimisation. The network is configured with several parameters, and during run-time network performance attributes are measured. These attributes are combined to a single value using a utility function that is to be maximised

## 2 Background and Related Work

as adaptation goal. Both parameters and utility value are fed into a knowledge base and new parameters are selected by simulated annealing. For this reason, the quality can also deteriorate in the mean-time, which is not acceptable in some situations. Also, the approach focuses on parameterisation of the network stack only.

Adaptation performed in a distributed fashion inside of the network uses simpler methods. For example, the Fennec Fox framework [SGC13] allows to switch between different configurations of the network stack during run-time based on events. Re-configuration is controlled by small Swift Fox scripts that explicitly define the different configurations with their algorithms and parameterisations, event conditions on sensor measurements and timer values that are evaluated periodically, and the transition between configurations based on these events. However, Fennec Fox does not support the user in finding the optimal configurations for different operating conditions and is focused on the network stack only.

In order to cope with the changing condition of its environment Impala [LM03] provides an interface for on-the-fly application adaptation. Several of these applications are installed on a common core that serves as event filter. Application parameters (e.g. number of neighbouring animals, amount of sensor data) and system parameters (e.g. battery level, geographic position) define the run-time state of these applications. Impala gathers these parameters regularly and, using a Finite State Machine, decides which application should be used. However, Impala's Finite State Machine has to be created manually by the user and the adaptation at application level is quite coarse grained.

In Chi [FET<sup>+</sup>10], configuration policies consist of code that is invoked when a parameter is changed (e.g. the number of current connections) and that in turn can change other parameters stored in a blackboard from where they are read by the algorithms and the application. However, these parameterisations need to be hard-coded and no system support is given to find them.

CONFab [AMM<sup>+</sup>12] automates the component based composition of the WSN protocol stack. Using detailed descriptions of the single components and of the scenario, possible protocol stacks are generated. Then, these stacks are evaluated and filtered based on user-defined goals, experiences from previous deployments and expert inputs, stored in a Knowledge Base. This process is formally represented as optimisation task. For unknown component combinations, CONFab is able to estimate their performance based on learned past deployments. For applications with different sub-scenarios, CONFab can create protocol stacks with alternative implementations that can be switched by a decision making component. However, this component needs to be implemented manually by the programmer.

Table 2.1 summarises and compares the discussed (re-)configuration systems. Only few systems support the user in finding good configurations and none of them features a distributed and dynamic adaptation process that is automatically created.

	Paradigm	Assessment of Configurations	Adaptation Process	Adaptation Triggers	Adaptation Specification	Adaptation Method
SPIN	distributed	no (single algorithm)	simple decision on energy level	energy level	hard coded in algorithm	switch between 2 fixed versions
PalChaudhuri	distributed	no (single algorithm)	fixed formula to derive parameters	application	hard coded in algorithm	change of parameters
SNACK	centralised	no	no adaptation	–	–	–
PCOM	distributed	no (properties of components given by user)	dependency resolution	new/vanished devices	static application contracts	component reselection
FiGaRo	distributed	no	user chooses target and new module	user	dynamic target declaration	exchange of modules
GRATIS(plus)/DESERT	centralised	no (properties of components given by user)	solving constraint satisfaction problem	QoS parameters	(static?) user constraints	switching of components
pTunes	centralised	theoretical model for algorithms	solving mixed-integer nonlinear program	network state	static user requirements	change of parameters
Meshkova	centralised	experimental at run-time	simulated annealing	network performance	(static?) utility function	change of parameters
Fennec Fox	distributed	no	finite state machine	local events	static scripts with FSM def.	switching of components
Impala	distributed	no	finite state machine	application/system parameters	static FSM definition	switching of applications
Chi	distributed	no	code invoked when parameters change	any parameter on blackboard	static C code	change of parameters
CONFab	distributed	reasoning on knowledge base before install	manual decision making component	any observable by dec. making comp.	static user goals	switching of components

Table 2.1: Comparison of application (re-)configuration systems

### 2.3.2 Code Exchange

Since reconfiguration that goes beyond the adjustment of parameters requires to change the installed program, code exchange strategies are reviewed now.

Interpreted scripts usually have a small size and, thus, they can be efficiently transmitted and stored in RAM. SensorWare [BHS03] defines a high-level language based on Tel with a radio, sensor, timer and mobility API. Mobile control scripts written in that SensorWare scripting language distribute themselves from node to node and can perform typical monitoring and aggregation functionality. However, resource-rich platform is required for execution. A virtual machine for sensor networks is implemented with Maté [LC02]. However, it has to be programmed in a low-level, assembler-like language, which makes it difficult for larger programs. A general problem of scripting languages is that the overhead to interpret them at run-time is considerable, especially on resource-constrained platforms.

Several approaches exist to provide exchange of native running code during run-time. By default, Contiki supports dynamic loading and unloading of modules [DFEV06] by dynamic linking at run-time. In contrast, TinyOS only offers to replace the complete code image using Deluge [HC04]. Although Deluge and also the following solutions were originally designed for code updates over the network, e.g. to fix errors, they can also be used to replace (parts of) the code to support run-time adaptation. In this case, the new code is already stored on the nodes, usually in the serial flash. FlexCup [MGL<sup>+</sup>06] takes advantage of nesC's binary components, which can be compiled separately, and introduces dynamic linking for TinyOS. However, this requires considerable effort in terms of time and energy as well as access to the external flash. TinyModules [GMN09] minimises the linking costs for TinyOS by defining a more static memory layout of the binary components and fixed entry points for all functions. Unfortunately, this required modifications in the compiler, which is hard to maintain. Moreover, tool support is minimal and the developer has to determine manually all external dependencies of the binary components and has to create new interface components and their wiring for all external connections. Dynamic TinyOS [MALW10] solves these problems partially by providing a compiler extension that helps in isolating the components and creating the separate binaries. Unfortunately, no further details are given how this is achieved.

All solutions for TinyOS discussed so far require a hardware reboot of the node. ELON [DLW<sup>+</sup>10] executes a software reboot by only initialising the updated components. However, this might cause conflicts since most other components are not prepared for it. ELON allows to create a single replaceable component by annotating all interfaces or functions that are to be called from the replaceable part in the static core and vice versa. Since there is no tool support for this, it seems to be feasible only for small components. No linking needs to be done on the node since the layout of the base application is known and special jump tables have been introduced. On TelosB nodes, the updated component can be held in RAM if the component is small enough, thus

completely avoiding Flash writes. However, this possibility does not exist for MicaZ's Harvard architecture.

QDiff [BSAH12] is one of many examples that mainly try to reduce the size of the code update and indirectly also try to minimise reprogramming. It compares the old C code and binary with the new one, identifies clones and differences and applies large reordering and rewriting to the new binary. A patch file is generated with insert, update, copy, pad and repeat commands, which is applied on the node. While this effectively reduces flash reprogramming it cannot avoid it completely. What is more, the update scripts can only be used to move from a specific code state to another code state. It is not clear if the update scripts can be created in such a way the transition between alternative algorithms in arbitrary order is possible. Certainly, update scripts would be necessary for all possible transitions. Otherwise, intermediate transitions would be necessary, which introduces unnecessary flash writes. QDiff also pretends to avoid reboots, but it does not explain how to initialise completely new application parts.

Independent of the operating system, all systems for replacing native code require reprogramming of the flash memory, consuming time and energy. One possible solution is to combine different algorithms into a single new one and to add intelligence to decide on the proper code segment to use. For example, the Multi-MAF routing algorithm [FNL09] includes a proactive scheme based on routing trees and a reactive scheme that starts disseminating the data using flooding. The system proposed by Krishna [KMO08] switches between algorithms of different complexity that segment foreground objects in camera images in order to save computational resources. However, the complexity to create such combined algorithms and to ensure their correctness can be very high.

In the MultiMAC framework [DNF<sup>+</sup>05] several MAC protocols are installed at the same time. When sending, MultiMAC chooses the one with best performance, and received frames are forwarded to the correct MAC protocol for decoding. However, MultiMAC runs on Linux systems, targets MAC algorithms only and requires certain interactions with the MAC protocols for correct operation.

More flexible solutions have been proposed that allow to switch between modularised algorithms during run-time. All three systems were already mentioned in Section 2.3.1 about (re-)configuration. Here, we focus on their code exchange part.

In Fennec Fox [SGC13], alternative algorithms for each layer of the protocol stack are installed in parallel and function calls are directed by switch statement to the enabled algorithms. However, the Fennec Fox framework is limited to a single component in each of the MAC, network and application layer. Also, all components need to comply with interfaces defined by the framework, which is, therefore, not flexible enough.

CONFab [AMM<sup>+</sup>12] is able to compare graphs of different constructed protocols stacks and can combine them by introducing a decision making component to create an adaptive application. However, this works only if the components are modelled with

their MAC Protocol Design MAC-PD [SAZM10], which results in a detailed XML model of the protocol. No details are given how these two alternative versions of the same protocol are included in the program at the same time.

GRATISplus [KNE<sup>+</sup>04] also proposes the use of switching components to support reconfiguration in TinyOS. However, it leaves their implementation completely to the programmer. The type of reconfigurable components is not limited; the components only need to implement TinyOS' StdControl interface and provide clean start and stop.

The discussed code exchanged systems are summarised and compared in Table 2.2. No generic system exists that avoids flash reprogramming and instead switches components, automatically creates such switchable components and is not bound to specific algorithm types.

### 2.3.3 Monitoring Systems

Most monitoring systems are used for debugging, fault detection and visualisation of the network state for an end-user. LiveNet [CPMW08] uses passive sniffer nodes to record every packet to local flash or to pass them to the serial port or an attached host where they are transmitted to a central unit via a backchannel. There, all traces are merged and then analysed to determine the behaviour of the network, e.g. coverage, hotspots, connectivity or path interference. In contrast, in Sympathy [RCK<sup>+</sup>05] operational nodes themselves collect internal information like routing tables or transmitted and received packets and send this information to the base station using the application's routing layer, which interferes with the normal operation of the network.

Some adaptive systems collect the monitoring information at one node, usually the base station, and take the adaptation there. pTunes [ZFM<sup>+</sup>12], which was already mentioned before, collects network state information such as topology and link quality using Glossy network floods [FZTS11]. Although this can be performed efficiently, the base station is a single point of failure and the system, thus, has issues with network partitioning.

Deng [DZL<sup>+</sup>10] analyses the computation of the minima and maxima in wireless sensor networks, but focuses on single-hop and clustered networks, which limits the applicability. Zhao et al. [ZGE03] present a monitoring framework that collects information in the network and distributes it to the whole network in an energy-efficient manner. However, depending on the actual implementation either the extreme values are only stable at the end of an aggregation period, which is sub-optimal to drive adaptation, or minimum/maximum values cannot increase/decrease fast enough when they become invalid.

The different monitoring systems are summarised and compared in Table 2.3. Most available systems are intended for debugging purposes and, thus, follow a central



	Exchange Method	Granularity	User Tasks	Reboot	Write Access
SensorWare	mobile control scripts	complete scripts	none	no	RAM
Maté	scripts for VM	complete scripts	none	no	RAM
Contiki	dynamic linking	single module	none	no	Progr. Flash
Deluge	code exchange	complete binary	none	yes	Progr. Flash
FlexCup	dynamic linking	single component	manual component creation	yes	Progr. Flash
TinyModules	dynamic linking	single component	manual component creation	yes	Progr. Flash
Dynamic TinyOS	dynamic linking	single component	none?	yes?	Progr. Flash
ELON	link at base	max 1 component	manual component creation	only reinit	Progr. Flash on MicaZ, RAM on TelosB
QDiff	patching installed code	any change	none	no	Progr. Flash
Multi-MAF	switching	none (single system)	–	no	none
Krishna	switching	none (single system)	–	no	FPGA
MultiMAC	switching	different MAC comp.	comply with MultiMAC interf.	no	none
Fennec Fox	switching	single protocol stack comp.	comply with Fennec Fox interf.	no	none
CONFab	switching	single protocol stack comp.	model components	no	none
GRATISplus	switching	single component	manual switching component	no	none

Table 2.2: Comparison of code exchange systems

## 2 Background and Related Work

	Paradigm	Aggregation	Infrastructure
LiveNet	central	none	sniffer nodes
Sympathy	central	none	none
pTunes	central	none	none
Deng	clustered	yes	none
Zhao	distributed	yes	none

Table 2.3: Comparison of monitoring systems

paradigm. Only Zhao’s approach supports distributed data aggregation for monitoring. Our work is based on this approach, but extends it to achieve a stable maximum value, which can also decrease fast. Furthermore, we integrate the monitoring system into our bigger adaptation framework TinyAdapt to drive the adaptation process.

### 2.3.4 Mobility Metrics

Mobility metrics have been widely studied in MANETs as instruments to predict link reliability for routing and to evaluate protocols. Xu [XBJ07] presents a theoretical analysis framework for well-known link and path metrics. The link/path availability is defined as the probability that a link/path that existed at time 0 also exists at time  $t$ . The link/path persistence is more strict since it requires that the link/path was continuously existing between time 0 and  $t$ . The link/path residual time estimates how long a link/path will continue to exist before it is broken. Finally, the link/path duration measures how long the link/path exists before it breaks.

Both Boleng [BNC02] and Qin [QK06] studied metrics to enable adaptive MANET protocols and evaluated link duration (see before) and link change rate. The latter is the number of new and/or broken links during a time period. While [QK06] found the link change rate useful as a mobility metric for adjusting the routing behaviour, [BNC02] found it less reliable than the link duration metric.

Contact-based metrics are evaluated by Khelil [KMR05]. A contact is defined here as a list of single encounters between two nodes, i.e. if a link between two nodes was temporarily lost during the observation period this counts as a single contact with two encounters. The encounter rate is similar to the link change rate since it measures the number of new encounters per time period. Similar, the contact rate denotes the number of contacts per time period. The number of encounters per number of contacts results in the encounter frequency. The encounter duration is the same as the link duration metric, while the contact duration is the sum of all encounter lengths per number of contacts. Son [SMSA14] extends this and presents the Path Encounter Rate PER that sums the single average encounter rates along a path.

However, all path-based metrics are not general enough since established routing paths are not available in all cases. The availability, persistence and residual time metrics

include probability estimations operating on theoretical mobility models, but are hard to compute in real-world scenarios. For all contact based metrics all encounters need to be stored, which requires a larger amount of main memory. This is also the case with all metrics that are based on time to store timestamps. Therefore, they should be avoided in WSNs.

Since nodes can also move in groups, group based metrics are established by Benmansour [BM11]. These metrics take into account the velocities and directions of a node and its neighbours and predict the future positions of the nodes. In the publication, this metric is used for cluster head election. The improvement of the cluster based LEACH protocol by Kumar is based on a mobility metric that takes into account the distances between the neighbours [KVPJ08]. However, the applicability of these metrics and other MANET metrics depends on the information available on the sensor nodes and the effort to compute them. Therefore, not all of these metrics are feasible for our general adaptation framework.

	Metric Object	Measured Value
path/link availability (Xu)	path/link	probability
path/link persistence (Xu)	path/link	probability
path/link residual time (Xu)	path/link	time/probability
path duration (Xu)	path/link	time
link duration (Xu, Boleng, Qin)		
encounter duration (Khelil)		
link change rate (Boleng, Qin)	link	availability
encounter rate (Khelil)		
node degree (Qin)	link	availability
contact rate (Khelil)	link	availability list
encounter frequency (Khelil)	link	availability
contact (loss) duration (Khelil)	link	time list
path encounter rate (Son)	path	availability
group mobility metric (Benmansour)	neighbourhood	availability, velocity, direction
mobility factor (Kumar)	neighbourhood	location

Table 2.4: Comparison of mobility metrics

In general, metrics in WSNs are used only to drive a single task like the mentioned cluster head election or quite often for the selection of promising routing paths. Our intention is to find a more general metric that characterises mobility as such in order to drive the generic adaptation. Table 2.4 summarises and compares the different mobility metrics. As explained before, most of them have issues with computability or memory overhead or require information that is not generally available. Although “link change rate” and “node degree” are general enough and can be computed efficiently, there is still room for improvement as will be shown in Section 4.4.

## 2.4 Conclusion

In this chapter, the special characteristics of Wireless Sensor Networks and their operating systems and algorithms have been shown. While operating systems provide basic abstractions and management of system resources, the TinyCubus project aims at reducing the complexity of WSN by offering self-configuration, cross-layer interaction, code exchange and adaptation. We have shown that existing reconfiguration frameworks, code exchange methods for TinyOS, and network monitoring systems have problems to reach these goals. This motivates our work for TinyAdapt and TinySwitch.

# 3

## The Generic Adaptation Framework TinyAdapt

Algorithms for Wireless Sensor Networks are usually designed for certain network conditions as already discussed in Section 2.1.3. Additionally, the algorithms try to optimise certain performance criteria, e.g. power consumption or delivery ratio for a routing algorithm. It depends both on the application scenario and the wishes and needs of the application's user which algorithms can be used for this application.

In long-running WSNs both network conditions and user requirements can change over time. The engineers that have installed a WSN could now change the software running on the nodes using one of the systems presented in Section 2.3.2. However, such manual changes imply high effort. Also, the engineers might not be available all the time, and end-users typically do not have deep knowledge of the technical details of an application.

It can also be the case that an application scenario constantly exhibits dynamic behaviour such as changes in network topology or in the traffic pattern. For example, in a logistics application goods are first moved around (mobile network) and need to be tracked while a simple observation of their status (e.g. cooling) is enough when they are stored in the warehouse or in containers (static network). Algorithms optimised for one distinct network state will not succeed here. This motivates the need for an autonomous adaptation solution.

In this chapter, the generic adaptation framework TinyAdapt is developed. As envisioned in the TinyCubus concept (see Section 2.2), it can autonomously choose appropriate algorithms and parameterisations based on current network characteristics and user requirements. First, we will discuss the meaning of adaptation, why it is needed and how it can be achieved. Then, we establish design principles that guided the development of the workflow to create an adaptive application and of the TinyAdapt architecture, which are both shown afterwards.

### 3.1 Adaptation

Adaptation is a central term for TinyAdapt. Therefore, this section provides a deeper look at the characteristics of adaptation and why and how it is performed in computer

systems in general and in sensor networks in particular. It concludes with feasible adaptation methods for *TinyAdapt*.

#### 3.1.1 Characteristics of Adaptation

Several definitions of adaptation can be found in different areas. In music or literature, adaptation means the editing or reworking of a literary or musical work, e.g. to change the genre or the instruments. This is done by other musicians or authors manually. The adaptation of the eye is an example from biology: the size of the pupil is increased or decreased based on the intensity of light so that the receptors are stimulated optimally. In computer systems, adaptation is the adjustment of hardware, software, or data to changed state in hardware, software, data, or the environment. This will be explained later.

In all examples, adaptation **changes the concrete appearance of a system, but leaves the core untouched**. The theme of the music, the story of the book, the function of the eye or the computer system does not change in principle. Secondly, the adaptation is **triggered due to a detected change in conditions**, either internal or external to the adaptive systems. Also in the musical example, the new need for the same piece with another instrumentation can be regarded as such a change. And thirdly, it **happens goal-oriented and strives for an improvement** of the system under the new conditions. The measurement of this improvement is not an easy task. In some cases, it can only be expressed in an ordinal scale, e.g. arrangement 1 of the music piece ‘sounds better’ than arrangement 2 which is quite subjective and cannot be assessed automatically.

It is possible to regard the biological evolution of a physiological process or an anatomical structure over time driven by natural selection as adaptation. In our definition, it is not: over a long time period, evolution alters also the core of a being. Single changes happen by accident and are not goal-oriented so that they can also lead to a deterioration. Thus, several points contradict our definition.

#### 3.1.2 Adaptation in Computer Systems

A small summary on adaptation in computer systems was already included in the previous section. In the following, we will describe the triggers for and the methods of adaptation in more detail. Thereby, we abstract from the mechanisms leading to the changes and carrying out the adaptation steps but concentrate on the actual changes that are made to the system and that are subsequently needed by the system.

As outlined before, several parts of a computer system can be adapted. Hardware can be exchanged as a whole (e.g. a faster processor) or a component’s characteristic can be changed by configuration parameters (e.g. the send signal strength of the WiFi adapter). Also, the whole software can be replaced, e.g. with an updated version

of the software, or it can be tuned by changing parameters. And last but not least, data can be changed, e.g. the level of detail of a presentation or its mode (text or graphic).

The triggers for adaptation are also manifold: The change of hardware components (e.g. adding or removing of main memory) or of the whole platform can require and trigger adaptation of the software to be able to work with the new hardware. If software is changed, data might need to be converted to other formats. On the other hand, if the format of the input data is changed, the software has to be adapted by installing filters that are able to understand the new format. Frequently, context is a reason for adaptation. Context is generally defined as “any information that can be used to characterise the situation of an entity” [Dey01], here the computer system. Examples are location, number and type of other devices, the type of environment of the computer system, and many more. For example, a mobile phone might automatically switch off the ring tone if the user is in a meeting. The distinction between data and context is not easy since the context can be stored as data in the network. But there can be other data, e.g. cartographic material, which is stored by the developer on the computers directly and is not acquired automatically.

When comparing the triggers and the adaptation possibilities, one notices that there are overlappings. For this reason, one adaptation step might activate another trigger. This can lead to problems when such a series of adaptation forms a loop. We consider each parameter either as input or output of the adaptation and possible dependencies have to be solved before performing actual adaptation.

### 3.1.3 Adaptation in Wireless Sensor Networks

In this section we evaluate whether the presented adaptation possibilities and triggers are relevant for Wireless Sensor Networks. This will also establish a border between sensor networks and close-by research areas.

Since a larger number of sensor nodes is deployed in many WSN scenarios it is unlikely that the user will manually change the hardware of every sensor node. Apart from approaches using robots, a WSN is also not able to do this autonomously — although it might solve many problems we have to deal with, e.g. concerning energy. This could lead to the assumption that the resource restrictions of sensor nodes will trigger adaptation. However, the resource restrictions do not change over time but are a general characteristic of WSNs, i.e. they are not a trigger but the reason for adaptation. Therefore, we do not have to consider hardware as a trigger or adaptation possibility. On the other hand, it is feasible to change the characteristics of the hardware by setting parameters, e.g. the signal strength of the radio module.

Usually, a WSN is run or maintained by a single user or a group of users who also have developed the software running on the nodes. It is unlikely that these users change one part of the software in such a way that another part is not working any more without

### 3 The Generic Adaptation Framework *TinyAdapt*

automatic adaptation. Rather, they will try to write and maintain an application of a piece. Thus, software is not a trigger for adaptation.

Regarding software as the medium of adaptation, we have to distinguish between the application and the supporting algorithms. The deployment and execution of new applications, which includes the detection of available resources and services on different computers and the assignment of functionality to them to run the complete application, is an important question and in the focus of other research projects and partly covered by TinyCubus' Topology Manager (see Section 2.2). *TinyAdapt*'s approach is orthogonal and assumes already an assignment of the tasks to devices but tackles the re-configuration of the installed services in order to keep performance.

Therefore, the adaptation of supporting algorithms is more promising: The system offers several services (routing, aggregation, time synchronisation, ...) to the application. It has been discussed in Section 2.1.3 that algorithms implementing these services depend on the context of the network. So, there is a clear need to adapt them when the network changes (see below).

For data as the source of adaptation, the same as for software applies: developers write and change their WSN software as a whole and, thus, it is unlikely that suddenly unknown data needs to be processed. On the other hand, application data might be adapted. For example, the quality and thereby the size of a data stream can be adapted using on-the-fly conversion to meet the load characteristics of the data path. However, the data is output by an algorithm running in the system and, therefore, one should be able to adapt the data by adapting the algorithms.

Finally, the environment of a sensor network can be a trigger for adaptation. Algorithms, e.g. for routing or time synchronisation, are mainly influenced by the "low level system context", i.e. other parameters like the number of neighbouring nodes or the mobility. In contrast, a lot of information from the environment is of little worth for our intended adaptation of supporting algorithms. The location or the type of the environment is mainly interesting for the application, e.g. to start a navigation system if the car moves out of town. This type of context dependency is handled by several Ubiquitous Computing projects.

To sum up, we will use certain information from the context to trigger the adaptation of supporting algorithms for the application software. We also identified hardware parameters as adaptable, but since hardware cannot be regarded as independent from these algorithms we will not manage its parameters directly but use the algorithms as an intermediate step so that they will set the appropriate parameters.

#### 3.1.4 Adaptation Possibilities

We have identified supporting algorithms as adaptable. The actual adaptation can be performed in two ways: parameterisation and exchange.



It might be the case that a single algorithm exists for different scenarios, e.g. for static and mobile environments, but needs a different parameter set. Then, it is the task of the adaptation process to set these parameters. Adaptation by parameterisation requires that the algorithm can be tuned by changing the values of parameters during run-time. Therefore, the algorithm must be prepared for the adaptation process by providing an interface to set these parameters.

The other type of adaptation is the complete exchange of one algorithm by another if they offer the same functionality but with different characteristics (e.g. two routing algorithms, but the first works in a static, the second in a mobile environment). Since the developer usually knows the different operating conditions of the WSN different algorithms can be provided already during deployment. The task of the adaptation process is to select and activate the appropriate algorithm(s).

To activate an algorithm it needs to be installed first. Section 2.3.2 showed some approaches to exchange (parts of) the running code on sensor nodes. These approaches usually involve reprogramming of the program flash memory (see also Section 6.7.3 for an evaluation of reprogramming costs). In contrast, all needed algorithms could be installed at the same time and a software framework switches between them by enabling only one algorithm and protecting all disabled algorithms from being executed.

	Parameterisation	Switching	Installation
Generality	Low	Medium	High
Speed	High	Low/Medium	Low
Memory Consumption	Low	High	Low/Medium

Table 3.1: Comparison of adaptation methods

Table 3.1 compares the three adaptation methods. *Parameterisation* is the most restricted method since it can only adjust algorithm within their given bounds, while *switching* can select between several pre-installed ones. This is also true for *installation*, but new algorithms could also be requested over the network, which makes *installation* even more general. Setting parameters is quite fast for *parameterisation*. Software *switching* could be similarly fast, but since most algorithms imply a startup time a medium speed is more likely. In contrast, the reprogramming task for *installation* of new algorithms is time consuming as will be shown in Section 6.7.3. *Switching* consumes a lot of memory since more than one algorithm has to be kept in memory at the same time in contrast to *parameterisation* and *installation*. Some *installation* approaches need some temporary memory to perform their installation task.

## 3.2 Design Principles

There are many possibilities to develop adaptive solutions for wireless sensor networks and other resource-constrained devices. Some existing solutions have been shown in

Section 2.3.1. However, these solutions do not completely meet our requirements. To guide the development of our solution, TinyAdapt, we first establish three main design principles:

1. **Ease of Use:** When developing applications the programmer often has the choice to develop everything from scratch and tailored to the specific application, or to build on existing frameworks and use their tools and services. To make such frameworks appealing to the programmer their use has to be easy, i.e. there should be **minimal extra manual work** involved and **minimal knowledge** should be needed. At the same time, the framework should provide **necessary services** whose use brings relief to the programmer. In summary, the savings here should by far outweigh the extra work needed for using the framework. If an adaptation framework like TinyAdapt follows this principle, new adaptive algorithms can then be developed more rapidly and existing algorithms can be converted to adaptive versions easily.
2. **Generic Applicability:** TinyAdapt should impose as little requirements as possible both on hardware and software. First, TinyAdapt should run on **standard sensor node hardware**. It should neither require one or more powerful devices in the network to make adaptation decisions, nor should it rely on a powerful base station or external computer, which represent a single point of failure and might even not be available in all scenarios. Second, TinyAdapt should be an open framework that **does not restrict the types of algorithms** than it can handle. Third, as explained in Section 3.1 adaptation can be triggered by general system context and can be performed in different ways. The framework should not restrict these **adaptation triggers and methods**.
3. **Effectiveness:** Adaptation is performed to re-establish application performance in changing system contexts (see Section 3.1). In doing so, adaptation strives for improvements, and the ultimate goal would be to even **achieve optimal performance**. On no account, adaptation decisions should lead to deteriorated systems. The decisions need to be taken in a **deterministic fashion** in order to be predictable so that adaptive systems could also be used in safety critical systems.

These design principles influenced the taken design decisions. We will refer back to them when presenting the workflow and the architecture in the following sections.

## 3.3 Definitions

The following terminology will be used throughout the description of TinyAdapt:

**Algorithm:** Algorithms are the basic entities of TinyAdapt. They provide a concrete service or protocol, e.g. the routing protocol CTP or the time synchronisation protocol RBS. Although the implementation of an algorithm can be split into

several components TinyAdapt only allows to configure (i.e., exchange or parameterise) an algorithm as a whole.

**Algorithm Class:** Multiple algorithms can provide the same type of function, for example routing or time synchronisation. This is called algorithm class. All algorithms of the same class are interchangeable in TinyAdapt, i.e. they can be exchanged by each other to perform adaptation.

**Algorithm Parameters:** The behaviour of a parameterisable algorithm can be adapted by changing its algorithm parameters  $A = \{A_1, A_2, \dots, A_a\}$ . These parameters are algorithm specific and although some can have the same name (e.g. beacon interval) their meaning can be different for different algorithms (e.g. because the unit is different).

**Performance Metrics:** The performance metrics  $P = \{P_1, P_2, \dots, P_p\}$  are a measure for the quality of a certain application configuration. While some performance metrics are general for all applications, e.g. power consumption, others only make sense if algorithms of a certain class are included in the application, e.g. delivery ratio for routing algorithms.

**Network Metrics:** The network metrics  $N = \{N_1, N_2, \dots, N_n\}$  are a measure for the network conditions. Examples of the network metrics include node mobility and network density. They are characteristics of the application scenario.

## 3.4 Workflow

TinyAdapt does not only consist of a software framework but also provides a workflow to guide the different actors involved in creating an adaptive application. This results from the *ease of use* principle since the development of such applications involves many steps that could otherwise only be performed by very experienced users. This workflow will be presented in this section.

During the search for optimal software configurations, various algorithms and parameters are usually tried out. This implies that system performance could also deteriorate after adaptation until this behaviour is detected and another adaptation step is executed. This would waste resources and could lead to non-deterministic behaviour, violating both the *generic applicability* and the *effectiveness* principle. Moreover, the search for better configurations can be quite complex, for example reasoning on knowledge bases or solving nonlinear programs. Again, the *generic applicability* principle is violated since such tasks usually cannot be executed on resource-constrained devices but need infrastructure support.

For these reasons, the TinyAdapt workflow starts with a **pre-installation phase** where the performance of different application configurations under specific network conditions is thoroughly explored. The best performing configurations are stored in a table from where a light-weight adaptation process selects a suitable entry during the

**run-time phase.** The disadvantage of this approach is that only known algorithms and parameter combinations can be used for adaptation and the system performance might be below the optimum for unknown conditions. This can be mitigated by a more fine-grained parameter exploration with realistic models or testbeds in the pre-installation phase. On the other hand, this approach ensures that only known and well-performing application configurations are used.

Figure 3.1 shows the TinyAdapt workflow in detail. The pre-installation phase can be divided further into an **algorithm modification** part and a **framework configuration and algorithm exploration** part. In the run-time phase, only the **run-time adaptation** happens with respect to TinyAdapt. These parts are also motivated by the different actors mainly driving the single steps. The following subsections will detail the tasks of these actors.

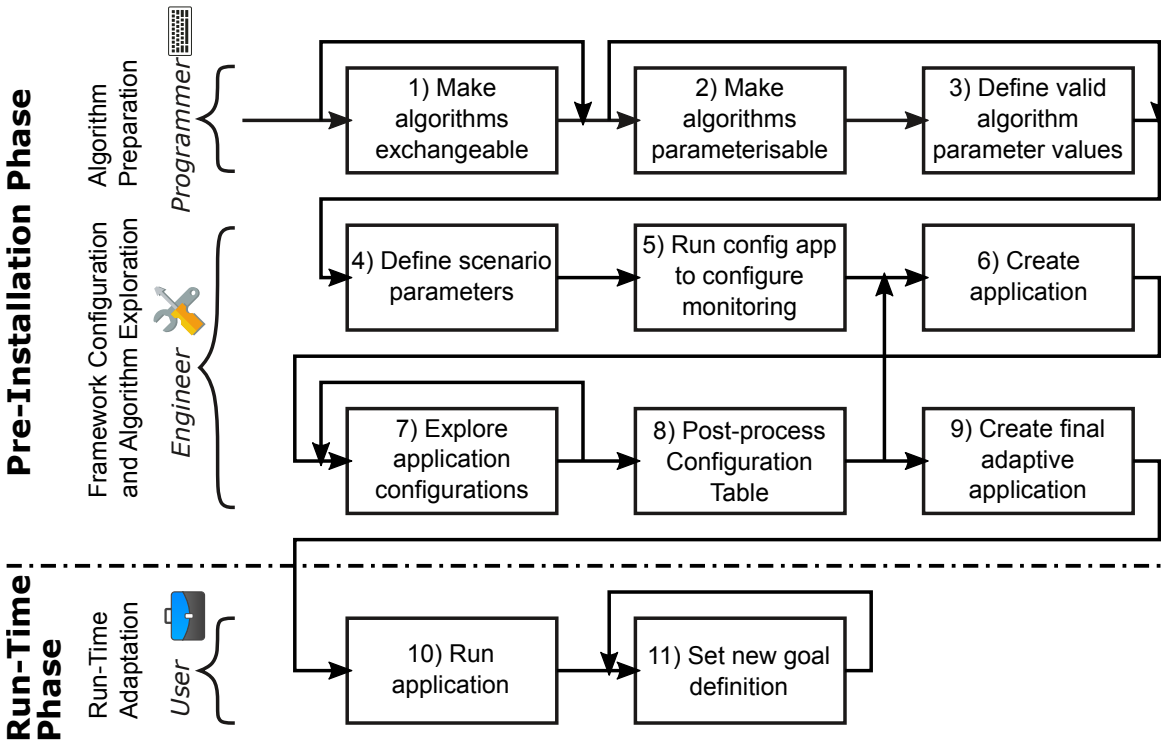


Figure 3.1: The TinyAdapt workflow and involved actors

### 3.4.1 Algorithm Preparation

In this first part of the workflow, algorithms have to be made compatible with the employed adaptation methods. In Section 3.1, code exchange and parameterisation have been identified as possible methods. If only a single algorithm for each algorithm class is used no code exchange is necessary and, thus, the task to make them exchangeable can be skipped. In contrast, if an algorithm cannot be tuned by parameters but can only be exchanged the task to make it parameterisable can be skipped. Of course,

an algorithm has to be made compatible with at least one adaptation method. Note that these modifications are necessary for any adaptation system and, thus, are not an extra effort of TinyAdapt compared to other adaptation systems.

As shown in Section 2.3.2 different code exchange systems exist, and in Chapter 5 we will present a new solution to switch between algorithms installed in parallel. TinyAdapt defines a simple interface towards these code exchange systems but does not directly interact with the exchangeable algorithms. Therefore, the necessary manual modifications by the programmer during step 1 depend on the type of used code exchange system. As rule of thumb, it is usually necessary to unify the interfaces of the algorithms, either only the application programming interface or also including all dependencies of the algorithm. If the code exchange system does not require a node restart or complete re-initialisation, the programmer should also make sure that each algorithm stops and re-starts cleanly, e.g. frees all resources or stops all internal threads when it is stopped by the adaptation process. For our TinySwitch solution, details on these modifications can be found in Chapter 5.

The algorithms that are adaptable by parameterisation need to share their algorithm parameters with TinyAdapt. TinyCubus proposes a central state repository where TinyAdapt sets the parameter values and the algorithms read the currently valid parameterisation to configure themselves. This minimises code dependencies between the algorithm modules and the TinyAdapt framework modules since no explicit wiring is required between them. Examples for suitable state repositories in wireless sensor networks are the cross-layer framework TinyXXL [LMM<sup>+</sup>06] or Chi [FET<sup>+</sup>10]. Again, manual modification of the algorithm is required by the programmer (step 2).

Note that the modifications in step 1 and step 2 are the only requirements to the algorithms. This makes TinyAdapt a general framework, fulfilling the *general applicability* principle. Since the necessary modifications might require a deeper knowledge of the algorithm they should be done by experienced programmers.

Usually, a parameterisable algorithm has been tested previously by its developer and only a few useful parameterisations have been identified in this testing process. Such parameterisations, for example, can be looked up in research publications for the algorithm. These values, combinations of values, or even a range for reasonable parameter values need to be determined in step 3 and provided to TinyAdapt as metadata included in the algorithm description.

The preparation of the algorithms can be done independently of a concrete application and scenario. Therefore, a universal algorithm library can be established; and later on when the engineer starts a new application adaptable algorithms can be selected. Although each algorithm only needs to support one of both adaptation methods it is obvious that a non-exchangeable algorithm cannot be an alternative for an exchangeable algorithm of the same class. Therefore, it is advisable that code exchange is always foreseen for algorithms in a generic algorithm library.

#### 3.4.2 Framework Configuration and Algorithm Exploration

The exploration of the different algorithms and/or their different parameterisations should assess how these algorithms perform with the final application under real operating conditions. Therefore, both the application and the testing scenarios need to be as close to the final application and conditions as possible.

The exploration can be performed through a simulator, a testbed or even a real deployment whereby the latter are usually closer to reality and, thus, better. Depending on the exploration method different scenario parameters need to be defined in step 4. When simulation is used, several simulator settings have to be provided, for example the number of nodes, their locations and, if required, a mobility model or sensor sample data. Similar settings are also applicable for testbeds. Although a real deployment is usually the best approach it can be difficult to achieve reproducible exploration runs, which can make it difficult to get several samples for each algorithm and parameterisation under test.

Different network conditions require separate exploration. For example, a real-world application could have a static phase and a mobile phase. In this case, different exploration scenarios should cover only one of these phases to clearly assess the performance of the algorithms under specific conditions.

Before the actual exploration, in step 5 a configuration tool provided by *TinyAdapt* needs to be run to configure the monitoring system including the network metrics (see Section 4.2). In short, this ensures that the network metrics are correctly computed and efficiently distributed. It is necessary to run this tool for all scenarios to ensure that monitoring is working under all network conditions.

Concerning the application, the best results are achieved when the final application is already used during exploration. Therefore, the engineer should create this final application already in step 6. If different algorithms should be tested with the application *TinyAdapt* must be able to select the algorithms automatically when building the application for exploration. For this purpose, *TinyAdapt* passes the names of the selected algorithms as parameters to a custom build script, which is provided by the engineer, where they need to be handled appropriately. For example, the names could be passed as symbols to the preprocessor or compiler where they are evaluated in conditional compilation directives. Alternatively, the build script could create a temporary version of the original source code and replace a placeholder with the name of the actual algorithm. Algorithm parameters  $A$  are passed to the build script in the same way, but their values can be set as simple constants.

The actual exploration of the algorithms and their parameterisation (step 7) is largely automated, but also time consuming. *TinyAdapt* will decide which configurations should be tested (see Section 4.5). The engineer can inspect the exploration results and instruct *TinyAdapt* to add more detailed explorations.

TinyAdapt will build a new application with the selected algorithm(s) and parameter(s) using the custom build script mentioned before. It will hand over the application binary to a custom exploration script that needs to be tailored by the engineer to the used exploration method. For example, the simulator is invoked or the nodes of a testbed are flashed with the binary. Also, the output data of the exploration run needs to be gathered, post-processed and finally returned to TinyAdapt as performance metrics  $P$ . For example, to calculate the delivery ratio, debugging output of the single nodes have to be collected to compute the total number of packets sent, which is compared to the total number of received packets obtained from base station output. TinyAdapt components that are already included in this explorative application observe the network conditions, which also need to be collected and returned to TinyAdapt as network metrics  $N$ . The delivery and gathering mechanism for  $P$  and  $N$  depends on the exploration method: in simulators all measurements can simply be printed; most testbeds have out-of-band delivery mechanisms, e.g. USB connections to the testbed infrastructure; in real deployments measurements can be stored locally and be transferred to the base station after the exploration, which is, e.g., supported by TinyLTS [SSFM11].

The results of all exploration runs are collected in the Configuration Table (CT). In order to make efficient use of limited memory of the nodes (see principle *general applicability*), the size of the CT must be reduced in a CT post-processing step (8). Thus, in this phase, the CT is examined to filter out irrelevant table entries through a reduction process described in Section 4.6. This step also tests the gathered network metric values if they are characteristic for a specific scenario that needs different algorithms or parameterisations. This either results in a further reduction of the CT or in a warning that the network metrics are not suitable to distinguish between these scenarios. In the second case, the engineer needs to refine the network metrics and repeat the exploration steps.

The final task of the engineer is to build the adaptive application (step 9) that includes all needed algorithms and the TinyAdapt run-time components. For our concrete TinyOS implementation of TinyAdapt, which we show in the next chapter, the complete run-time system is included with a single line, which needs to be added to the main configuration. Depending on the method for code exchange, additional changes could be necessary to include the Installation module. For example, when using TinySwitch (see Chapter 5) instead of including a specific algorithm the generated multiplexing layer is used.

The initial adaptation configuration, the so-called Goal Definition (see next section), also needs to be included in the final application to enable adaptation at all. Also, when the initial scenario is known the correct initial algorithm configuration can already be determined offline and pre-set for the application to avoid immediate adaptation. Finally, the engineer is usually also responsible for installing the application and to deploy the nodes.

#### 3.4.3 Run-time Adaptation

This workflow part belongs to the run-time phase. It is the operational phase of the application (step 10) and TinyAdapt only works in the background.

In this phase, user interaction may not be possible in all envisioned scenarios, for example in long-term monitoring systems, which operate unattended, or in embedded systems, which are not visible to the user at all. Even if end-users are present they might not have knowledge about the system internals and the meaning of single parameters. Therefore, the framework has to make its adaptation decisions autonomously (see *ease of use* principle).

Different ways for representing the adaptation specification have been mentioned in Section 2.3.1. Some of them have already been excluded since they cannot be evaluated on normal sensor nodes (e.g. solving constraint satisfaction problems). To evaluate the outcome of an adaptation decision, utility functions are often used that map several measured performance metrics to a single value. The problem is that even simple goals require the user to create a complicated multi-attribute utility function [KC03], violating again the *ease of use* principle. Moreover, such complex functions are often hard-coded since their evaluation in an interpreted way is too costly for resource-constrained devices. This makes it difficult to change the adaptation goal during run-time.

We argue that the user usually has certain minimal requirements based on performance metrics that need to be fulfilled, and on top of that certain preferences for optimisations. If the minimal requirements cannot be assured the user wants to control in detail which requirement should be weakened. Therefore, TinyAdapt uses a more explicit and intuitive definition of user preferences, the so-called **Goal Definition**, to specify requirements as well as relaxation and optimisation strategies. Table 3.2 shows an example of such a Goal Definition. The **requirements** define the optimal performance that the user of the application requests and that the running system has to fulfil. They can be expressed as an inequality with the metrics of  $P$ . Experienced users can also define constraints on algorithm parameters  $A$  if they want to restrict the possible solution space. Since the system might not be able to meet these requirements under all operating conditions (e.g. it might not be possible to achieve a high delivery ratio in high mobility situations) the user can specify with **relaxation** rules how to weaken which requirement in which steps until a lower or upper bound is reached. A requirement can appear multiple times in the relaxation rules. This way the user has detailed control how the requirements are loosened and which of them are more important. Finally, **optimisations** guide the system to choose a single table entry if more than one is available after applying the (possibly relaxed) requirements. This adaptation process is described in detail in Section 4.7.

During installation (step 9) the initial goal definition was already installed on the nodes. However, the requirements of the user can change during the lifetime of the application. Therefore, the user can re-define the adaptation objective by setting



Requirements	1. delivery ratio $\geq 90\%$ 2. energy $\leq 200\text{J}$
Relaxation	1. decrease requirement 1 in steps of 10 until 70 2. increase requirement 2 in steps of 100 until 400 3. decrease requirement 1 in steps of 10 until 50
Optimisation	minimise energy

Table 3.2: Example Goal Definition

new Goal Definitions (step 11). TinyAdapt will monitor changes both in the Goal Definitions and in the network metrics and perform adaptation accordingly to meet the requirements of the Goal Definition under current network conditions.

In case the network changes largely or other unexpected behaviour occurs that leads to an unacceptable deterioration of performance the workflow can be repeated from the beginning. A code exchange strategy can be used to update the complete framework including the algorithms to also cope with such changes.

Note that component based systems allow for application composition by specifying the type of needed services, e.g. that routing is needed or that a certain output device is needed, and the system tries to fulfil these functional requirements (see Section 2.3.1). Instead of reorganising the whole application, our framework concentrates on the effective and efficient adaptation using a set of previously explored algorithms. Nevertheless, the selection of these algorithms could be performed by such component systems during the pre-installation phase. Hence, these approaches are orthogonal.

## 3.5 Architecture and Data Flow

While the previous section has shown the necessary steps to work with TinyAdapt, this section explains how TinyAdapt itself is built. Since a two-stage approach with a pre-installation and a run-time phase is adopted the architecture does not only cover the software components to build an adaptive application but it also encompasses the connections between the two phases. These connections are made of data generated during the pre-installation phase and used in the run-time phase. Figure 3.2 shows the architecture of the TinyAdapt framework and the overall data flow. For both phases, the structure of the application is shown, consisting of the operating system, the main program, its supporting algorithms and several framework components.

Adaptive algorithms and applications share some common functionality: they need to monitor some system conditions they want to react on, need to decide if they should react to changes, and finally need to alter their operation. This common functionality should be provided by an adaptation framework, removing this task from the user (see *ease of use* principle). This observation already proposes four of the main framework components: Monitoring, Adaptation Engine, Installation and Settings.

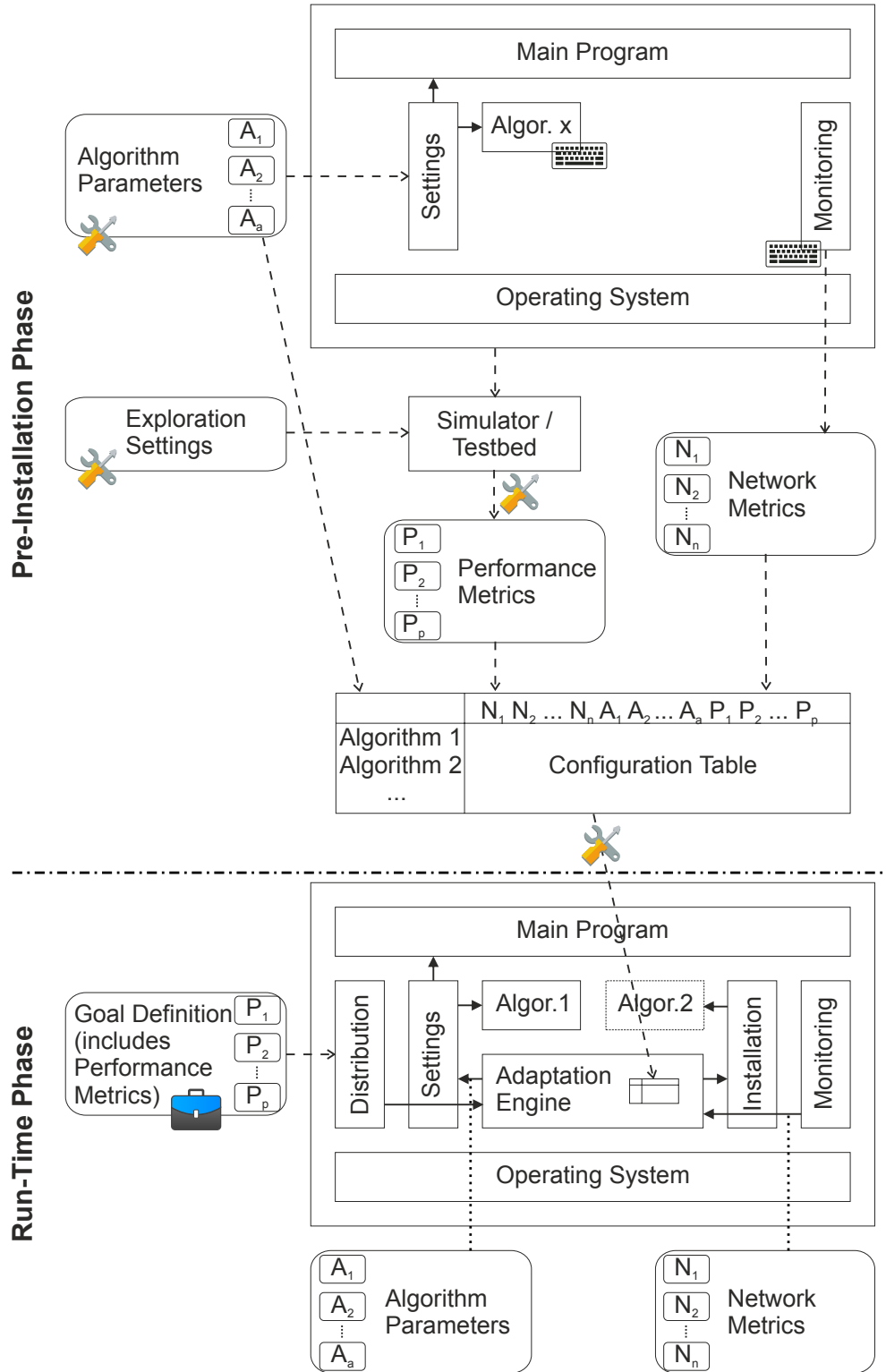


Figure 3.2: The TinyAdapt framework architecture and data flow

Data flow inside the application, which is relevant for TinyAdapt, is indicated by solid lines. Dotted lines that are connected to solid lines give details about the type of data. Interactions and data that are external to the application and controlled by the overall framework are indicated with dashed lines. Measurements and settings involved in this data flow are shown as rectangles with round edges. Finally, the three actors are represented by the same icons as in Figure 3.1: a keyboard for the programmer, a wrench and screwdriver for the engineer, and a briefcase for the end user.

### 3.5.1 Pre-Installation Phase

In the pre-installation phase the different application configurations are explored, optionally also for different network conditions if this is indicated by the application scenario. For this purpose, the application is built with different algorithms and/or different parameterisations for these algorithms. Algorithms have to use TinyAdapt's Settings component, which is already included in this phase, to get current parameterisation  $A$ . The set of tested algorithm parameters is mainly chosen by TinyAdapt, but can be influenced by the engineer.

TinyAdapt also offers the possibility to set parameters of the main program. Typically, these parameters control the amount or interval of data generated or sent. Thus, parameterisations for low-energy sub-scenarios do not need to rely on lower-level algorithms to reduce or compress data but can already avoid data at the source.

There is no Installation component present in the pre-installation phase as the algorithms will not be exchanged during run-time. Instead, as described in the workflow, TinyAdapt will build a new static binary with a specific algorithm and parameter configuration.

The actual exploration of the different configurations can be performed through a simulator, a testbed or even a real deployment. These exploration tools need to be configured by the engineer to match the final deployment settings as close as possible. The exploration runs result in various performance metrics  $P$ .

The Monitoring component that is already part of the application during the pre-installation phase delivers the network metrics  $N$ . Although the exploration settings are controlled by the engineer it is necessary for the framework to measure the network metrics in the running application. The reason is that no general functional dependency between the exploration settings and the measured metrics can be established in general (see principle *general applicability*). Therefore, it is advisable to monitor the metrics needed to drive adaptation during run-time already in the same way during exploration in the pre-installation phase. The monitoring component is described in detail in Section 4.3.

The tested combinations of algorithms and their parameterisations  $A$ , the measured network metrics  $N$  and performance metrics  $P$  are gathered in the Configuration

### 3 The Generic Adaptation Framework TinyAdapt

Table. During a post-processing step, bad combinations, which will never be used for adaptation since other combinations are better, can be removed from the table.

It is important to note that the framework can handle arbitrary  $A$ ,  $N$  and  $P$  as long as their values are integers (see *general applicability* principle). Algorithm parameters are not interpreted at all by TinyAdapt and, thus, just values are passed. If an new algorithm class has to be explored with respect to a new performance metric  $P_p$  (e.g. a time synchronisation algorithm should be explored based on the synchronisation accuracy) this metric can also be added easily as long as the engineer can translate exploration output to a single meaningful integer that can be compared to other values of the same metric. The same requirements hold for network metrics  $N$ .

#### 3.5.2 Run-Time Phase

During run-time, the Monitoring component measures the current network metrics  $N$ . Also, the users can set and change their adaptation preferences through the Goal Definitions that are distributed by the Distribution component in the whole network. The adaptation engine observes changes in Goal Definitions and network metrics and looks up the best configuration in the Configuration Table. Finally, it uses the Installation module to exchange algorithms and/or the Settings module to change their parameterisations  $A$ .

The adaptation decision is taken autonomously on each sensor node and not communicated to other nodes. Instead, the Distribution and Monitoring components ensure that all nodes eventually use the same basis for their adaptation decision by delivering the same value for network metrics  $N$  and the same Goal Definition to all nodes. This approach has the advantage that also local characteristics can influence the adaptation, for example the role of a node assigned by the Topology Manager (see Section 2.2.2).

Basically, only the Monitoring and Adaptation Engine component are provided by TinyAdapt. For Distribution, Settings and Installation existing systems can be used that offer the needed functionality and have appropriate interfaces for TinyAdapt: the Distribution component distributes data in the whole network and notifies TinyAdapt of its availability; the Settings component stores data, which is set by TinyAdapt and queried by other components; and the Installation component exchanges code when instructed by TinyAdapt to do so. This make TinyAdapt largely general and usable in different systems, following the *general applicability* principle.

## 3.6 Conclusion

This chapter has developed the principles, workflow, architecture and data flow for the generic adaptation framework TinyAdapt after having analysed the general concept

of adaptation in detail. The framework guides the involved actors through the process, which includes the modification of the algorithms to be used, their exploration and finally the run-time adaptation. TinyAdapt takes over much of the tedious and repetitive work and involves the programmer, engineer and user only when necessary, as suggested by the *ease of use* principle.

Most of TinyAdapt's work is done in a pre-installation phase, resulting in a very light-weight run-time adaptation phase. Thus, TinyAdapt can efficiently run on resource-constrained systems such as WSNs. At the same time, TinyAdapt does not limit the types of algorithms, performance metrics and adaptation triggers and methods that can be used with TinyAdapt, thus fulfilling the *general applicability* principle.

The deterministic use of known application configurations should result in optimal application behaviour if the algorithm exploration is performed correctly. This was required by the *effectiveness* principle.

In this chapter, we have not provided concrete implementations for all TinyAdapt components yet. Therefore, the TinyAdapt concept is generic and can be implemented on different operating systems. In the next chapter, we will provide a TinyOS implementation to integrate it with TinyCubus.



# 4 TinyAdapt Implementation

While the previous chapter has presented the general architecture and data flow of TinyAdapt this chapter shows the concrete implementation of TinyAdapt for TinyOS. This includes the software components that will be included in the running application as well as automated processes performed by the framework to create an adaptive application. After an overview of the run-time components, this chapter follows the workflow (see Section 3.4) of TinyAdapt to describe the implementation of the framework.

Throughout this chapter we apply the described methods to a hypothetical data collection application using an (artificial) routing algorithm. The application sends regular measurement data to a base station. The user cares about the data delivery ratio and about the energy consumption of this application. There are two algorithm parameters that influence its performance: the beacon interval of the routing algorithm and the low-power listening (LPL) interval of the MAC layer (see [ML08]). The scenario has a static and a mobile phase, which need to be optimally supported by the algorithm. We show how to build an adaptive application using TinyAdapt.

To include TinyAdapt, the application just needs to include one component, which links all necessary internal components of TinyAdapt. This can be done in any configuration, but a good place is in the main configuration:

```
configuration SampleAppC {}  
implementation {  
    components TinyAdaptC;  
    ...  
}
```

## 4.1 Run-Time Modules

The general architecture shown in Figure 3.2 includes five components necessary for run-time adaptation in TinyAdapt: Distribution, Settings, Adaptation Engine, Installation and Monitoring. While some components have been newly developed, others have been taken from the TinyOS library or from the overall TinyCubus project.

### 4.1.1 Distribution Component

It is the task of the **Distribution** module to make the current Goal Definition available to all nodes in the sensor network and to inform the Adaptation Engine about updates. Such functionality is provided by Drip [LPCS04], a standard TinyOS component. With Drip, nodes (re-)broadcast values periodically with exponentially increasing time intervals, starting with 1s up to 1024s. This interval is reset to 1s when a new value is set to make it disseminate fast. Sequence numbers are used to make sure that all nodes eventually agree on the latest and same value. This also works with different providers for this value, i.e. several base stations can be used to control the behaviour of the adaptation through the dissemination of Goal Definitions, and Drip ensures that all nodes eventually agree on the same Goal Definition. Drip also informs TinyAdapt when a new Goal Definition is available so that the adaptation process can be started.

Base stations that are able to insert new Goal Definitions into the network need to include a second TinyAdapt component. Thus, their main configuration could look as follows:

```
configuration SampleAppBaseC {}
implementation {
    components TinyAdaptC;
    components TinyAdaptBaseC;
    ...
}
```

TinyAdaptBaseC adds a serial connection, receives new Goal Definitions via this connection and passes them to Drip, which in turn starts disseminating them.

### 4.1.2 Settings Component

Parameters of the algorithms and possibly also of the main program are set via the **Settings** component — either directly by the engineer during the pre-installation phase or by the Adaptation Engine during the run-time phase. For this purpose, we use TinyXXL since it was developed in the TinyCubus context.

A description of TinyXXL can be found in Section 2.2. We apply it now to the routing component in our sample application, which might have originally looked as follows:

```
module DataRouter {
    provides {
        interface Init;
        ...
    }
    uses {
        interface LowPowerListening as LPL;
    }
}
```



```

    interface Timer;
    ...
}
}
implementation {
    command error_t Init.init() {
        call LPL.setLocalWakeupInterval(250);
        call Timer.startPeriodic(60 * 1024);
        return SUCCESS;
    }

    void sendMsg() {
        call LPL.setRemoteWakeupInterval(&msg, 250);
        ...
    }
    ...
}

```

The local wake-up interval for LPL and the timer period are set at startup, and when sending a packet the (same) remote wake-up interval is set. To share these two intervals between the algorithm and TinyAdapt they have to be declared first:

```

xldata RoutingParam() {
    uint8_t BeaconInterval = DEFAULT_BEACON_INTERVAL;
    uint16_t LPLInterval = DEFAULT_LPL_INTERVAL;
}

```

The default settings for both parameters can then be set in the application's make file by adding:

```
CFLAGS += -DDEFAULT_BEACON_INTERVAL=60 -DDEFAULT_LPL_INTERVAL = 250
```

This feature can be used later on when TinyAdapt automatically creates several static applications to explore the various parameter combinations. The shared data is included in the DataRouter module by adding the following to its header:

```
uses xldata RoutingParam;
```

The existing code needs to be changed in such a way that it uses the shared data instead of the constants:

```

command error_t Init.init() {
    call LPL.setLocalWakeupInterval(RoutingParam.LPLInterval);
    call Timer.startPeriodic(1024uL * RoutingParam.BeaconInterval);
    return SUCCESS;
}

```

## 4 *TinyAdapt* Implementation

```
void sendMsg() {  
    call LPL.setRemoteWakeupInterval(&msg, RoutingParam.LPLInterval);  
    ...  
}
```

Finally, the algorithm needs to react on changes in these parameters. There is only a single notification event for all values of a `xldata` definition. However, this fits *TinyAdapt* perfectly since during adaptation all parameters are changed at once.

```
event void RoutingParam.dataChanged() {  
    call LPL.setLocalWakeupInterval(RoutingParam.LPLInterval);  
    call Timer.stop();  
    call Timer.startPeriodic(1024uL * RoutingParam.BeaconInterval);  
}
```

After these straightforward changes, the algorithm is ready for parameter-based adaptation by *TinyAdapt*.

### 4.1.3 Other Components

An **Installation** component is only necessary if more than one algorithm exists in an algorithm class and if it needs to be exchanged during run-time to perform adaptation. The *TinyCubus* project already provides *FlexCup* [MGL<sup>+</sup>06]. Due to the drawbacks of code exchange we developed a code switching alternative, *TinySwitch*, which is presented in Chapter 5. There, an explanation can be found how applications use this algorithm switching solution.

Also, the Adaptation Engine, Monitoring and associated metric modules were developed especially for *TinyAdapt* and are explained in the following sections.

## 4.2 Framework Configuration Application

In this step, important parameters of the *TinyAdapt* framework are determined that influence the monitoring results (see Section 4.3) and the mobility metric (if used; see Section 4.4). Therefore, they are essential for good adaptation results.

Configuration of the mobility metrics starts by estimating the transmission range of the nodes. In free space, this could be found analytically, but in real world obstacles typically limit the range. To get realistic values for testbeds and the final deployment, the engineer could install a network mapping application on a static network that records all connections between the nodes. This is often done anyway to ensure that the deployed sensor network is well connected. If the locations of the nodes are known, the transmission range can be determined from the network map. Conservative bounds are preferred since otherwise the error will propagate in the next step.

Knowing the transmission range, the beacon interval of the mobility metric can be determined using the formulas that will be presented in Section 4.4. Assuming that two nodes travel at maximum speed, their relative speed can be calculated and the packet interval can be set to a value for which the relative travelled distance during the packet interval is equal to twice the transmission range.

This packet rate (as inverse of the packet interval) is used as start value in the first test application that simply sends beacons with this rate and records the number of neighbours and the number of link changes. From these numbers, statistical data is calculated and used with the formulas of Section 4.4 to obtain the theoretical ranges for the final mobility metric with different smoothing factors of the exponential moving average. The engineer can then select a factor that achieves both a good separation of velocity classes and a low delay.

It is advisable to align the broadcast interval of the monitoring framework to the beacon interval of the mobility metric since the metric values can then be included in the beacons. To estimate the convergence time of the monitoring framework a second test application measures the flooding time using the given beacon interval. To do so, each node starts packets that are flooded to the network with increasing sequence numbers. Each node receiving a packet checks if the packet is new, puts new packets into a queue and records its time of arrival. At the next beacon interval, the delay that the packet experienced at the node is added to the packet's total delay and the packet is re-broadcasted. This abstracts from the transmission time, but in comparison to the beacon interval it can be neglected. The delays with which each packet reaches each node are collected and evaluated offline afterwards. The engineer can set the metric stabilisation time of the adaptation process to approximately the average flooding time.

This step will be evaluated in Section 6.2 and concrete calculations will be shown there. Therefore, we will not repeat it here for the sample application.

## 4.3 Monitoring

TinyAdapt is developed to restore optimal application performance in changing network condition. To be able to do so, the Adaptation Engine needs to be aware of the current network status. The monitoring component serves to generically provide this network status by consistently delivering the values of the network metric set  $N$  to all nodes in the network. It is integrated into the application in both the pre-installation and the run-time phase and performs the same task in these phases as explained in Section 3.5.1. The network metrics are computed by various metric plugins that can be connected to the monitoring component during compile time. This flexible plugin concept stems from the *general applicability* principle. Also, it increases the *ease of use* since the engineer only needs to provide the metric values and the framework takes care of the network wide distribution.

## 4 *TinyAdapt Implementation*

To perform uniform adaptation on all nodes, the adaptation decision must rely on the same values for the network parameters as explained in Section 3.5.2. Since each node can experience different dynamic network behaviour, its local metric values may be different. A network-wide average, minimum or maximum can characterise such different local values. However, the efficiency and applicability of the distributed calculation must be considered. [DBTV08] gives an overview of several distributed averaging algorithms. Unfortunately, synchronous algorithms cannot be used since all nodes need to be synchronised and are required to compute a new estimate in each iteration. On the other hand, asynchronous algorithms, which only require two nodes to exchange and adjust their current estimate, can take a long time to reach a common value.

Based on these reasons, our approach uses the network-wide maximum of the locally-observed network parameters to drive the adaptation. The computation of the maximum is the main task of the monitoring component. Although a single area can dominate the maximum, this higher measurement can still characterise the network as we will show in Section 6.2 for our mobility metric. To minimise the effect of temporary high values occurring for metric variables with high variance it is advised to smooth them locally before handing them over to the monitoring component. Moreover, the maximum is easy to compute and only incurs little overhead. Note that the same argumentation also holds for the minimum and our algorithm also works with the opposite comparison operators.

### 4.3.1 Computation of Network Wide Maximum

Figure 4.1 shows the pseudo-code for computing the network-wide maximum. Each node stores four values: its local value `localvalue`, the current network maximum `maxvalue` and its source ID `maxsource`, and the sequence number `sequenceno`. Information is exchanged by periodically broadcasting a monitoring message.

The sequence number is increased when a node detects that it can provide a new network wide maximum and afterwards regularly by this node to indicate freshness of the value. When receiving a monitoring message with a higher sequence number than the currently stored `sequenceno` a node updates `sequenceno` to the received sequence number. This starts a new round of maximum computation, and each node first assumes that it provides the maximum. Immediately afterwards and subsequently each time a monitoring message is received during the same round, each node compares its currently known `maxvalue` with the received maximum and uses the higher value and its corresponding source for further propagation. Since two nodes can provide the same maximum the node with the lower node ID is taken as source. This is necessary for the algorithm to stabilise and can be found twice in the algorithm.

Figure 4.2a shows an example network with 5 nodes and the stored `localvalue` 1, `maxvalue` `m`, `maxsource` `s` and `sequenceno` `n`. Node 5 currently provides the `maxvalue` 60 with `sequenceno` 4, but node 3 has just joined the network and thinks

```

when monitoring msg m is received:
  if m.sequenceno > sequenceno
    maxvalue = localvalue
    maxsource = NODE_ID
    sequenceno = m.sequenceno
    timeout = MONITOR_TIMEOUT
  end if
  if m.sequenceno == sequenceno
  and (m.maxvalue > maxvalue
    or (m.maxvalue == maxvalue and m.maxsource < maxsource))
    maxvalue = m.maxvalue
    maxsource = m.maxsource
    timeout = MONITOR_TIMEOUT
  end if

when local value is changed:
  if maxsource == NODE_ID or localvalue > maxvalue
  or (localvalue == maxvalue and NODE_ID < maxsource)
    setLocalValueAsNewMax()
  end if

when timer is fired:
  if --timeout == 0
    setLocalValueAsNewMax()
  end if
  broadcast monitoring msg(maxvalue, maxsource, sequenceno)

function setLocalValueAsNewMax()
  maxsource = NODE_ID
  maxvalue = localvalue
  timeout = MONITOR_TIMEOUT
  sequenceno++

```

Figure 4.1: Computation of network-wide maximum

#### 4 TinyAdapt Implementation

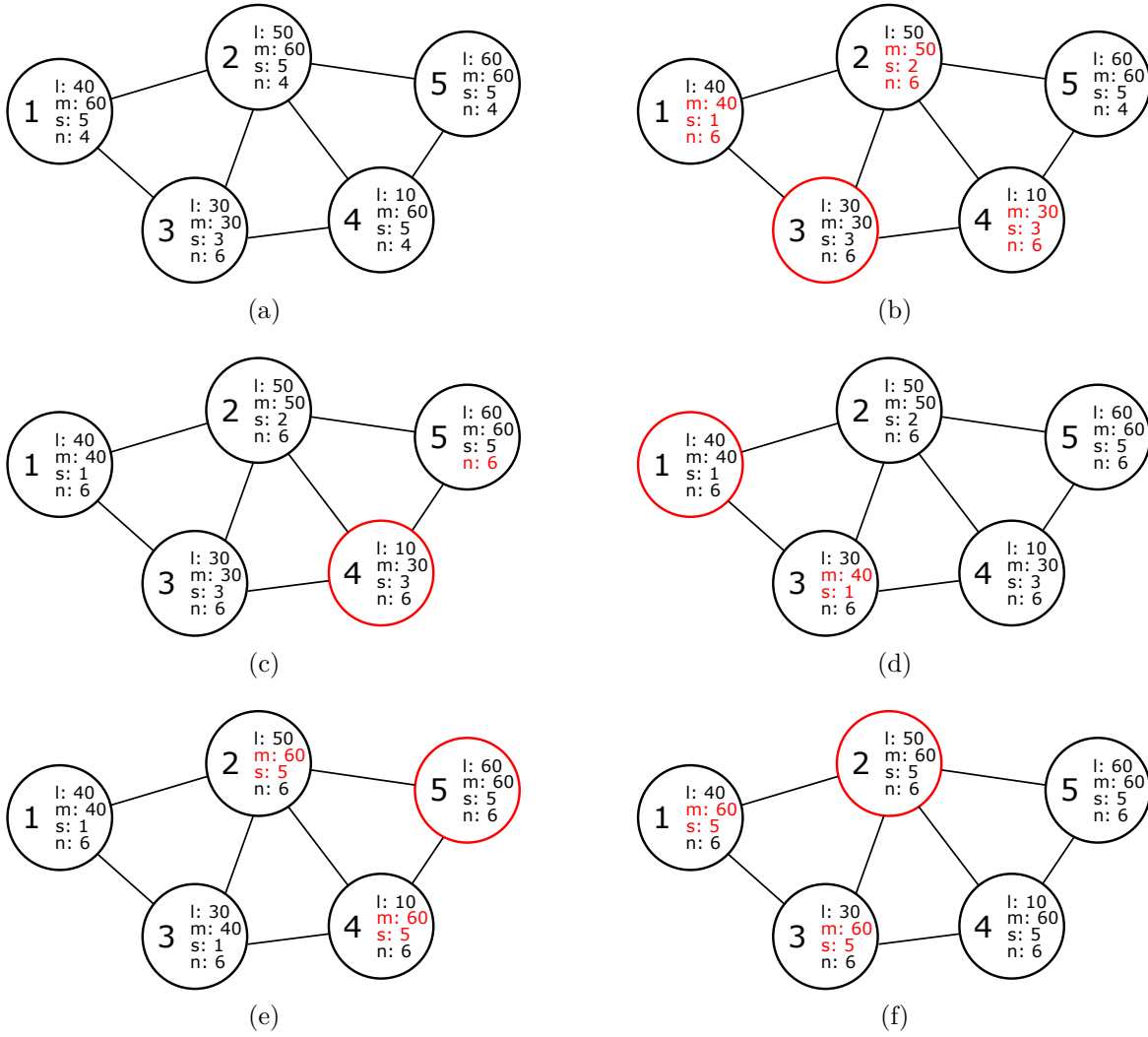


Figure 4.2: Calculation of network-wide maximum example: basic operation

it provides the maximum itself with **sequenceno** 6. Message transfer is asynchronous, and we assume that node 3 sends **m**, **s**, **n** first to its neighbours (Figure 4.2b). Nodes 1 and 2 set their **sequenceno** to the received value but detect that their **localvalue** is larger than the received **maxvalue** and, therefore, use their local data. Only node 4 uses the received data completely. Node 4 might be next to send, which only updates the **sequenceno** of node 5 (Figure 4.2c). In the mean time other nodes could also send their data, e.g. node 1, which would update other nodes, here node 3 (Figure 4.2d), but it will only increase the **maxvalue**. Finally, the provider of the real network-wide maximum (node 5) will broadcast its data (Figure 4.2e), which eventually reaches all nodes (Figure 4.2f).

When the local value changes and it is larger than the current network-wide maximum, it is set as the new maximum and a new round is started. Note that a special case can happen: if the current network-wide maximum is originated from the local node,

the new local value is set in all cases as the new maximum. Since this value will be propagated the maximum can also decrease until it will be eventually overwritten by a higher maximum of another node.

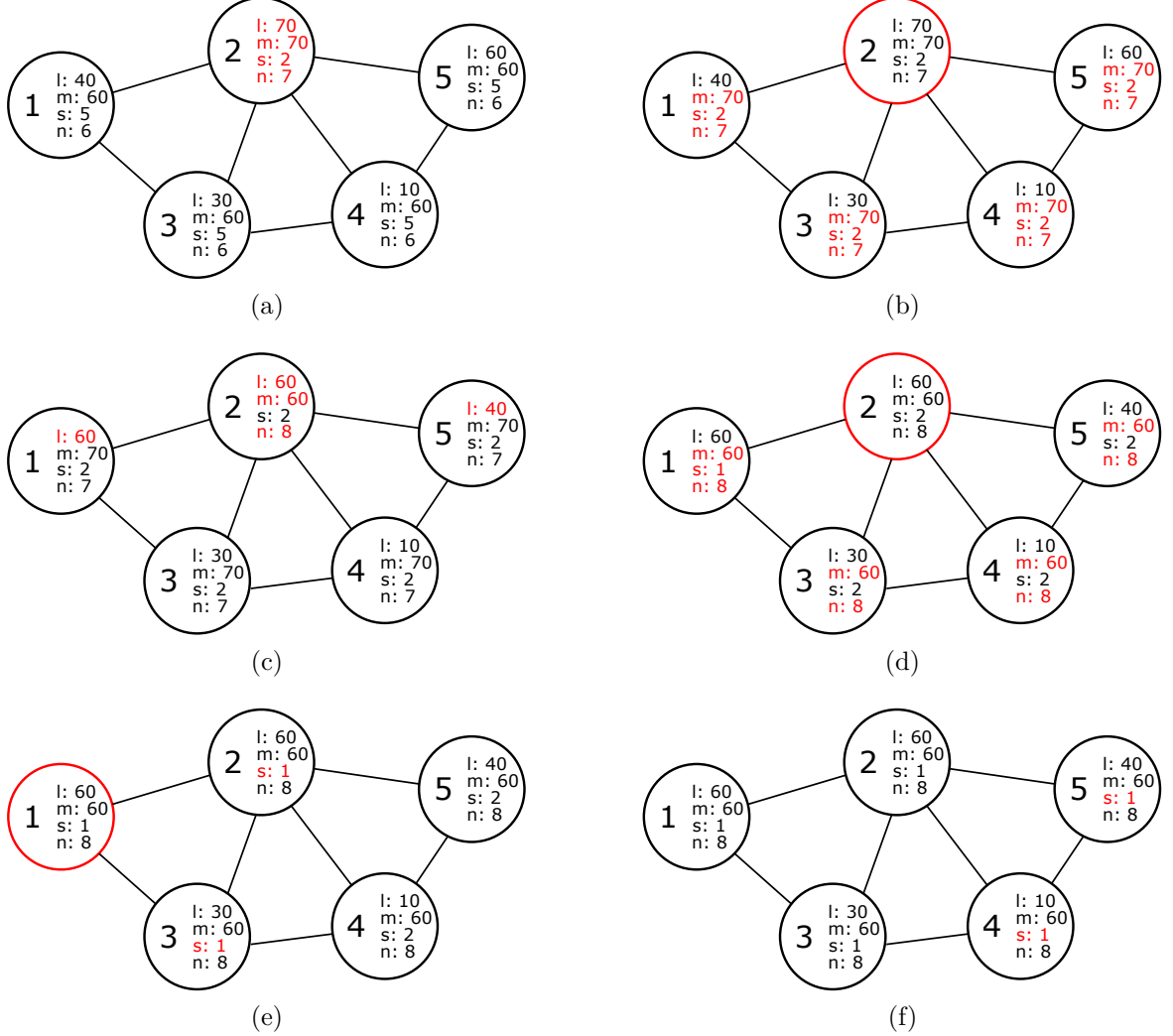


Figure 4.3: Calculation of network-wide maximum example: update of maximum

In our example, if the **localvalue** at node 2 changes to 70, node 2 assumes this as new global maximum and starts a new round by increasing the **sequenceno** (Figure 4.3a). It will send this data to its neighbours, which all accept it as new maximum (Figure 4.3b). Later on, the **localvalues** of node 1, 2 and 5 change to 60, 60 and 40, respectively. Since node 2 is the current provider of the network-wide maximum it starts a new round by increasing the **sequenceno** (Figure 4.3c). If the data of node 2 is received by its neighbours, nodes 3–5 take over this information. However, node 1 detects that it can provide the same value but has lower node ID. Therefore, it sets **maxsource** to 1 (Figure 4.3d). From there, the updated **maxsource** will propagate through the network (Figure 4.3e) until all nodes have the same data (Figure 4.3f).

#### 4 TinyAdapt Implementation

In addition, a timeout counter is decreased every time a node broadcasts its current maximum. It is reset when a new maximum is set or the current provider of the maximum increases the sequence number. If the timeout counter reaches zero it is assumed that the original provider of the maximum disappeared, and the local value is set as the new maximum. The timeout counter's default value is based on practical experience to tolerate temporary transmission problems; in our implementation we use 3 as default.

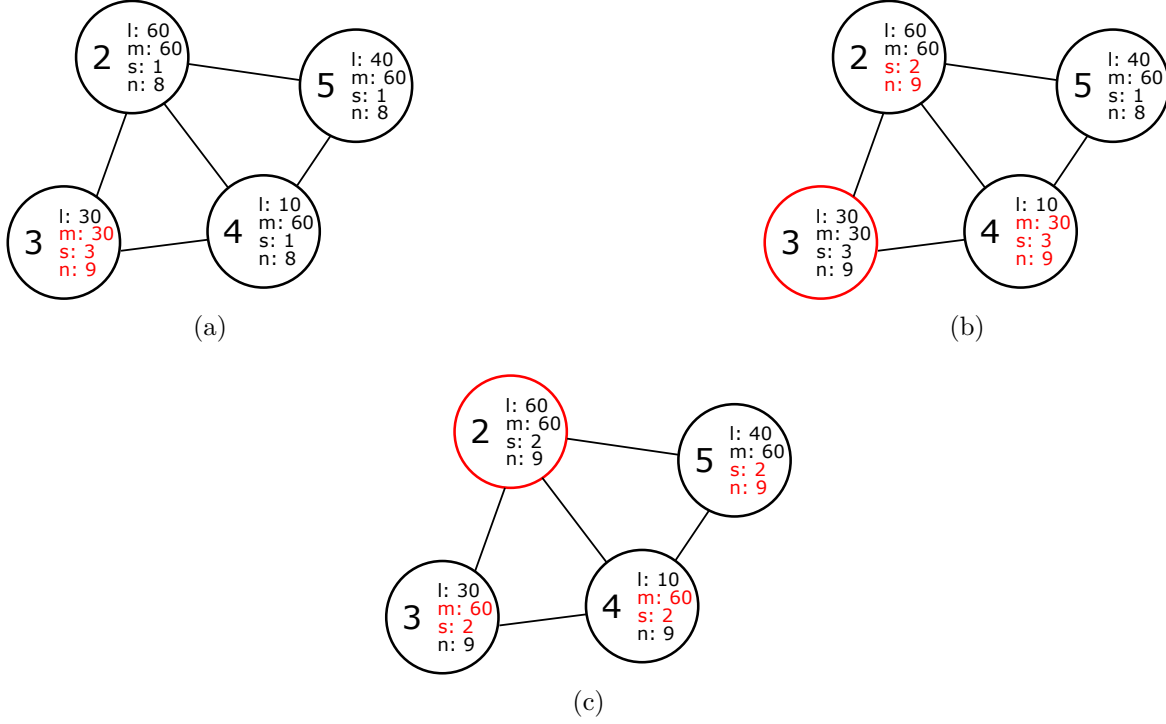


Figure 4.4: Calculation of network-wide maximum example: timeout

In our example, node 1 disappears. The timeout counter of node 3 is the first to reach 0. Therefore, node 3 starts a new round with its data as new maximum (Figure 4.4a). While node 4 takes over this information, node 2 detects that its **localvalue** is larger and, thus, sets this as new **maxvalue** (Figure 4.4b). If this data is finally transmitted to the neighbours all will again agree on the same network-wide maximum and its source node (Figure 4.4c).

The time until all nodes know the network-wide maximum depends on the broadcast interval of the monitoring message and the number of hops to reach all nodes via flooding from the node holding the maximum. In static networks, the latter is bound by the network diameter. However, in mobile networks the topology is always changing and partitions can occur as well. [CPS09] has shown that under certain conditions the flooding time decreases when the velocity of the nodes increases. In [CST11] they have shown how the flooding time is affected by different mobility patterns. However, asymptotically upper bounds only can be found from which it is hard to



derive practical values for our monitoring component in such a way that it achieves to disseminate the maximum value in most of the cases. Thus, a simple experiment provided by TinyAdapt shall be performed by the engineer (step 6 of Figure 3.1 and Section 4.2) to gather the average flooding time for a specific scenario.

With this algorithm, the maximum values of the network parameters can converge to the same values on all nodes and are delivered to the adaptation engine to achieve uniform network-wide adaptation.

### 4.3.2 Piggybacking

Since the monitoring protocol generates periodic monitoring messages to calculate the network-wide maximum the communication overhead should be reduced as much as possible. If the application or another algorithm sends packets anyway the monitoring data could be piggybacked, i.e. it could be added to the application or algorithm data and be sent along with it. Although this technique is well-known the data flow of the application and implementation details in TinyOS must be considered.

The effectiveness of piggybacking is determined by three parameters: the application data interval  $d$  in which packets are sent by the application, the piggybacking interval  $p$  in which on average data is generated that should be piggybacked, and the maximum waiting time  $w$ , which is the time that we can wait for an application packet until a dummy packet is created to carry the piggybacking data. Usually,  $w < p$  since there has to be enough time to create and send a dummy message before the new piggybacking interval starts. If  $d < w$  all piggybacking data should find an application packet. If  $d > w$  only  $w/d$  of the piggybacking data can be transported by application packets on average, when considering a single node. In this calculation we assume that all application packets have enough free space for the piggybacking data.

In a multi-hop network where all nodes generate data that is to be routed to a base station the probability of finding a packet to piggyback on increases depending on the distribution of the sending events during  $d$  and on the proximity of a node to the base station. This estimation, though, provides a lower bound, in reality higher savings can be achieved.

We show in the following how the piggybacking module is included in TinyOS and how it changes a network packet to add additional data.

#### TinyOS network stack

In TinyOS 2, all modules sending or receiving packets usually rely on four different components: `AMSenderC`, `AMReceiverC`, `AMSnooperC` and `AMSnoopingReceiverC` that provide the interfaces `AMSend` and `Receive`. Figure 4.5 shows how all these interfaces are finally wired to the `ActiveMessageC` component (solid rectangles and arrows only).

#### 4 TinyAdapt Implementation

This component is the first platform-specific component that wires to the communication components specific for the platform, for example `CC2420ActiveMessageC` or `CC1000ActiveMessageC`.

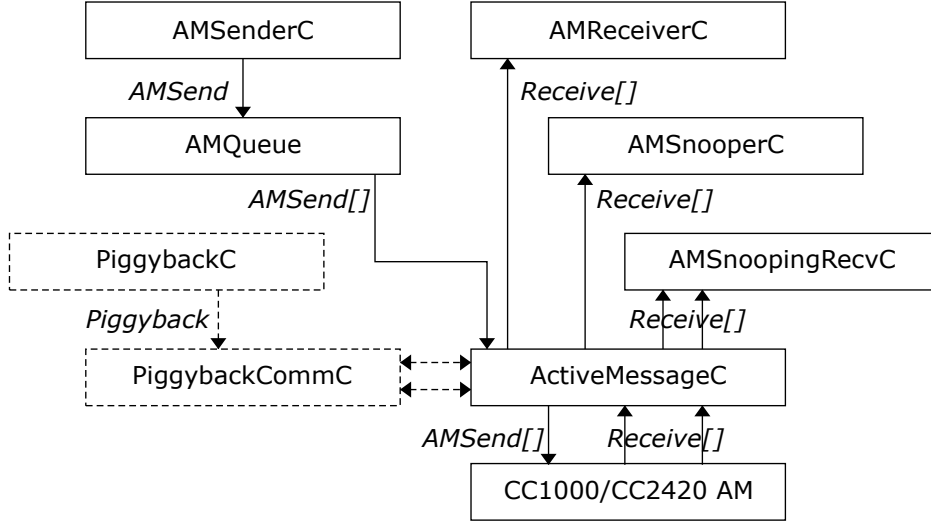


Figure 4.5: TinyOS network stack with Piggybacking

For a completely platform-independent implementation we would have to intercept the packets on top of `ActiveMessageC`. Sending is easy since all packets to be sent are routed through `AMQueue`, which can be replaced easily. Receiving data is more complicated since the parameterised `Receive` interface of `ActiveMessageC` is split into single AM-id specific interfaces by generic `AMReceiverC`/`AMSnooperC`/`AMSnoopingReceiverC` configurations. Therefore, the distinction between a piggybacked and non-piggybacked packet cannot be done by the AM-id, but an extra flag or field has to be added to the header which again makes the solution platform-dependent and, moreover, decreases the payload size even if no data is piggybacked.

It is also impossible to wire the complete parameterised `Receive` interface of `ActiveMessageC` to our own component, i.e. in parallel to `AMReceiverC`/`AMSnooperC`/`AMSnoopingReceiverC`. Although our own component would be called every time before the components that are wired through `AMReceiverC`/`AMSnooperC`/`AMSnoopingReceiverC` (at least in the current implementation of nesC 1.3.1), our piggybacking component can neither change the AM id nor the length of the data, which is necessary to hide the extra piggybacked data from the rest of the application.

Due to these TinyOS implementation reasons, the only way to implement a generic piggybacking component is to replace `ActiveMessageC`. Although we have to provide an implementation for each platform, this configuration is small and the required wiring is straight forward: only the `AMSend` and both `Receive` interfaces have to be rerouted through `PiggybackCommC`; the other wiring can remain unchanged.

### Piggybacking module

A module using our piggybacking module requests sending of a data within a certain time. When a packet is about to be sent by the application the piggybacking requests are checked sorted by urgency. A `readyToSend` event is signalled to the requesting module with the position in the packet payload and the space available. The requesting module can then append data to the application's data and returns the amount of this piggybacked data. The piggybacking module recalculates the available space and the new position in the packet payload. This continues until the packet is full or all requests have been handled. If the deadline of a piggybacking request expires the piggybacking module generates an empty dummy packet, which is processed in the same way as described to carry the outstanding data.

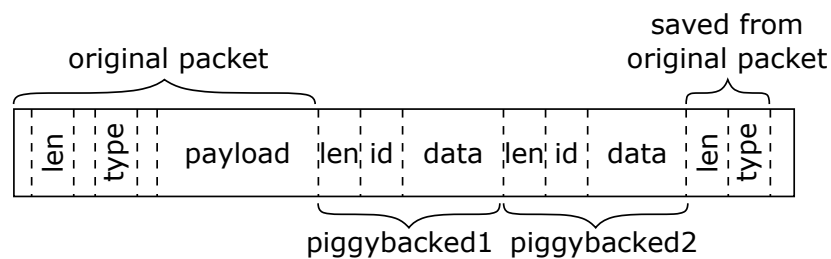


Figure 4.6: Format of a Piggyback packet

Figure 4.6 shows a packet with piggybacked data. After the original packet, the length of the piggybacked data and the id of the piggybacking channel are stored, followed by the actual data. As shown, a packet can contain several piggyback data payloads. To indicate that a packet contains piggybacked data the AM id is changed; the original AM id is saved together with the original length at the end of the packet. Then, the packet is returned to `ActiveMessageC` from where it is sent in the usual way.

When a piggyback packet is received, an event is signalled for each piggybacked data payload with the position of the data and the data length as parameters. After this, the original AM id and length are restored and the packet is processed further by `ActiveMessageC` as usual.

The piggybacking functionality can be used by including a new `PiggybackC` instance in a `nesC` configuration, each generating a new channel id. Several piggybacking channels can be used in parallel. For example, the monitoring component can calculate several extreme values for different metrics with different sending rates if the metrics differ in volatility.

## 4.4 Mobility Metric

Most WSN protocols can cope with different densities of nodes and also with small topology changes due to the addition and removal of nodes during run-time. However,

## 4 *TinyAdapt Implementation*

node mobility has a major and continuous impact on the network topology. WSN protocols typically assume either a static or a mobile network and are optimised for this condition since this allows for major energy savings. For example, the CTP routing algorithm [FGJ<sup>+</sup>07] builds and maintains a routing tree, which becomes unusable when nodes are moving. In contrast, when links of the tree are stable for a longer time the beacon interval for neighbour detection is increased exponentially, thus saving energy.

Due to this prominence of mobility for WSN operation mobility is a major adaptation driver. For example, with MS-MAC [PJ04], an adaptive mobility-aware MAC protocol has been developed that creates sleep/wake-up schedules more often when mobility is increasing. Therefore, we develop a suitable mobility metric for adaptation in this section. Nevertheless, TinyAdapt is of course generic to handle various dynamic network behaviours using different metrics that can be plugged into the monitoring component.

In [BNC02] a set of requirements for mobility metrics in MANETs is presented and they are valid for Wireless Sensor Networks as well. First, the metric has to be computed locally on the nodes. Therefore, no information only available to a simulator or other central instance can be used. Second, the computation has to consider the limited resources of sensor nodes, especially memory and energy. For this reason, systems like GPS cannot be used to calculate the velocity of a node. Third, the metric should be generic and not be related to a specific protocol. This is a major requirement for our adaptation framework since it should enable the adaptation of a wide range of algorithms and algorithm types.

Analysis done in Section 2.3.4 showed that most of the existing metrics violate at least one of these requirements. A cheap, yet powerful approach is to count changes in the neighbourhood, e.g the number of new and lost neighbours during a time interval (also called link change rate), which only requires a list of neighbours and a timeout counter. Moreover, the average number of neighbours in a time interval can be determined. While these metrics and values can be easily computed, the problem is that both are affected by the node density. It is obvious that with higher density a node has more neighbours and more new neighbourhood links get established and more old links get lost in the mobile case. Therefore, we devise a new metric by computing “link changes per number of neighbours”, which has only little dependency on the total number of nodes and is thus more suited for adaptation.

### **Determination of Beacon Interval**

The accuracy of all packet-based mobility models depends on the frequency of the neighbours’ beacons. If the application does not generate enough packets, dummy beacon packets have to be transmitted. However, due to the movement of the nodes a packet can still be missed. Although this results both in a lower link change rate and a lower number of neighbours, the latter is affected for a longer time depending

on the timeout interval and, thus, the value of the composed metric is higher. [AS07] calculates the probability of missing a contact as

$$P_M = \sin \theta_H + \frac{\pi - 2\theta_H - \sin 2\theta_H}{4 \cos \theta_H}$$

with  $\theta_H = \arccos(T_p \tilde{v}/2r)$  for  $0 < T_p \tilde{v} < 2r$ , where  $T_p$  is the packet interval,  $\tilde{v}$  is the relative speed of two mobile nodes and  $r$  is the transmission radius. When both nodes have the same speed,  $\tilde{v} \approx 4v/\pi$ . [AS07] assumes a unit disc graph radio model. When lossy models are applied  $r$  should be set to a value that ensures the delivery of a packet with very high probability; the real probability of missing a contact will be smaller. In case the relative travelled distance during the packet interval is equal to twice the transmission radius,  $\theta_H$  becomes 0 and  $P_M = \pi/4$ , i.e. the probability to miss this neighbour is 21.5%. This error can be acceptable if the metric values for different velocities are still distinguishable. With this knowledge, the engineer can set the minimum packet interval for the metric.

### Metric Calculation

It has been shown in [QK06] that the number of neighbours and the link change rate of a node vary over time. It is thus infeasible to use a single value at a certain point in time to drive adaptation. Therefore, an additional smoothing process should be performed. One approach is to compute the average over a moving window of values. However, this approach may require a lot of memory depending on the window size, and the variance of the resulting average can still be quite high since a single value could dominate the average.

A better alternative is to use an exponential moving average (EMA), which has also the property to predict short-term future values. EMA requires only the storage of the last value since it is calculated as  $x_t^* = \alpha x_t + (1 - \alpha)x_{t-1}^*$ , where the coefficient  $\alpha$  represents the degree of weighting decrease,  $0 \leq \alpha \leq 1$ ;  $x_t$  is the observation at a time period  $t$ ;  $x_{t-1}^*$  is the value of the EMA at a time period  $t - 1$ . Since both of the inputs for our final metric have high variance, we first compute an EMA of the number of neighbours  $n_t^*$  and of the link changes  $c_t^*$ . Then, the metric  $m_t$  is calculated as  $m_t = c_t^*/n_t^*$  and smoothed with the EMA as  $m_t^*$ . This final average is then passed to the monitoring component as the local value for the network-wide maximum.

Note that the value of  $\alpha$  is crucial for this process since on the one hand it reduces the variance of the metric, which is the desired property. On the other hand, the answer of the adaptation system will be delayed when the network conditions change since the EMA will reach  $qx_i$  at  $t = \frac{\log 1-q}{\log 1-\alpha}$  when a constant input signal  $x_i$  is applied. For example, with  $\alpha = 0.3$  the EMA reaches 95% of the actual value at  $t = 9$ .

## 4 TinyAdapt Implementation

[Hun86] has shown that the variance of the EMA can be calculated from the variance of the original data as  $\sigma_{EMA}^2 = \frac{\alpha}{2-\alpha}\sigma^2$ . Further, when calculating  $Q = X/Y$  of two random variables  $X$  and  $Y$ , the variance of  $Q$  can be estimated as

$$\sigma_Q^2 \approx \left(\frac{\mu_X}{\mu_Y}\right)^2 \left[ \left(\frac{\sigma_X}{\mu_X}\right)^2 + \left(\frac{\sigma_Y}{\mu_Y}\right)^2 - 2\rho \frac{\sigma_X \sigma_Y}{\mu_X \mu_Y} \right]$$

where  $\rho$  is the correlation coefficient for  $X$  and  $Y$ . Although for our metric the quotient  $Q$  is not calculated from the random variables but from their EMA, the correlation coefficient of the raw values is still a valid characteristic number. As argued above both our single measurements are affected in the same way by the total number of nodes. Therefore, the correlation coefficient is quite high and, subsequently, the variance of the final metric is much lower than the variance of the single metrics.

Using simple experiments for different velocities (step 6 of Figure 3.1), the engineer can collect the locally observed values for the number of neighbours and the number of link changes during each monitoring interval in the specific scenario. TinyAdapt calculates the mean  $\mu$  and variance  $\sigma^2$  of both as well as the correlation coefficient. Using the above formulas, the variance of the final metric can be predicted for different values of  $\alpha$ . Thus, the engineer can choose a value for  $\alpha$  so that both the reaction time is short and a large fraction of the metric values, e.g.  $\mu \pm 2\sigma \approx 95\%$ , can still be assigned to the correct velocity. Section 6.2 contains a concrete example for the experiments and the configuration of the measurements.

Similar questions need to be answered when other metrics for network dynamics are developed: which factors are influencing the metric by how much? To which extent can this be tolerated or how can it be mitigated? What is the variance of the locally observed values and how (e.g. smoothing) can it be reduced? Some of the techniques presented above could be applied if the metric is also based on packet detection, but more general recommendations are difficult since the metrics and their foundations can be quite diverse, especially when specific hardware is involved.

## 4.5 Algorithm Exploration

In the exploration step, the performance of algorithms and their parameterisations is measured, if present also in different scenarios. The number of possible parameter combinations seems to be quite large, but the algorithm programmer has already provided a range for the values or a specific list of useful values during the algorithm preparation step (see Section 3.4.1). The engineer can additionally define a minimal interval for each parameter, i.e. the difference between single values that are to be tested will not be lower than this interval.

TinyAdapt currently supports two exploration approaches: brute-force exploration and adaptive exploration. In brute-force exploration, all possible combinations of

parameter values, based on the given list or range and the minimal interval, are tested. Due to the possibly large number of combinations, this is useful only if there is little knowledge about the algorithm in general, and it is possible only for simulation on compute clusters with massive parallelisation. On the other hand, the engineer will get a good picture of the algorithm's behaviour and its configuration possibilities.

The adaptive exploration approach tries to optimise the search for good parameter combinations. However, common optimisation approaches use a single goal function for which a single optimum is to be found. In our case, several parameter combinations can be found that all result in maximal local performance. For example, when two metrics  $A$  and  $B$  with a range of  $[0, 10]$  (higher is better) are to be optimised, there could be two parameterisations resulting in a maximum performance for only one of the metrics, e.g.  $(9, 2)$  and  $(3, 8)$ . However, there could also be a parameterisation resulting in a trade-off between both metrics, e.g.  $(6, 5)$ . We aim at finding all of them to give the user free choice during run-time.

As input, the engineer provides the information if higher or lower values are preferable for each performance metric and which differences in the measurements are to be considered significant. Then, it will start the exploration at the boundaries of the parameter space, i.e. with combinations of the minimal and maximal values of the parameters. From these exploration results it selects those parameter combinations as favourites for the first round that maximise at least one performance metric. In each round, the favourites are examined further by dividing the parameter range for each parameter and testing the resulting combinations if they have not been tested before. A combination is considered as a favourite for the next round if the exploration results improve at least one metric significantly compared to its "parent combination". Search stops if the parameter range cannot be divided further without going below the given minimal interval.

When a new scenario is explored, the successful parameter combinations of the previous scenarios are also tested at the beginning. If scenarios are similar, this can speed up the exploration since the optimal parameterisations might also be similar. Moreover, it is necessary to test the same parameterisations in order to be able to merge scenario results in the post-processing step.

In our example application from the beginning of this chapter, the programmer has defined suitable values for the beacon interval (20, 40, 60) and the LPL interval (0, 250, 500). Performance is measured in terms of delivery ratio and energy, for which the significant difference is set to 5 and 20, respectively, and of course higher delivery ratio and lower energy is better. Since the application consists of a static and a mobile sub-scenario two exploration runs are necessary.

The adaptive exploration for the static scenario is shown in Table 4.1. In round 1, the boundaries of the parameter space are tested. The third (60;0) and the fourth (60;500) combination maximise delivery ratio or minimise energy. Therefore, they are our favourites and considered for round 2. From these favourites, the parameter range is divided: for (60;0) the child combinations are (40;0) and (60;250) and for (60;500)

#### 4 TinyAdapt Implementation

Round	$A_1$ Beacon Int.	$A_2$ LPL Int.	$P_1$ Deliv. Ratio	$P_2$ Energy
1	20	0	100	400
1	20	500	97	200
1	60	0	100	395
1	60	500	97	120
2	40	0	100	400
2	40	500	97	160
2	60	250	99	100
3	40	250	98	130

Table 4.1: Adaptive exploration of example application — static scenario. Favourites marked yellow

they are (40;500) and again (60;250). Only (60;250) shows significant improvement compared to its parents. Child combinations are (40;250), (60;0) and (60;500), but only the first one has not been tested yet. Results of round three show no significant improvement and since no further parameter space division is possible exploration ends.

Compared to the brute-force exploration approach only one combination did not have to be tested. The adaptive exploration shows its advantages when more values of a parameters are to be tested, i.e. when the interval is smaller. Therefore, Table 4.2 contains the results of the mobile scenario using brute-force exploration.

$A_1$ Beacon Int.	$A_2$ LPL Int.	$P_1$ Deliv. Ratio	$P_2$ Energy
20	0	77	400
20	250	72	310
20	500	71	315
40	0	66	400
40	250	62	250
40	500	55	305
60	0	56	400
60	250	47	200
60	500	41	250

Table 4.2: Brute-force exploration of example application — mobile scenario



## 4.6 Configuration Table Post-Processing

After the exploration step, the Configuration Table (CT) lists all tested algorithms with their parameterisation and the measured network metrics for the test case with respect to specific performance metrics. Depending on the exploration approach and the location of the local optima, the CT can contain many entries. Storing the raw values in the sensor nodes and using them as basis for adaptation would impose two major problems: the efficient storage of the whole CT in the memory of resource constrained devices and the efficient selection of entries that match the network metrics due to their continuous values. The first problem is solved by a reduction process and the merging of exploration results, the second by a mapping of the network metric values.

### 4.6.1 Removal of Inferior Entries

The combination of different algorithm parameter values and possible network conditions leads to a large number of scenarios that are recorded in the Configuration Table. This table could be kept in program flash memory of a sensor node, but each data access to the program memory incurs overhead on Harvard architectures and the table is frequently accessed during the adaptation process. Instead, storing the CT in data memory also gives the engineer the opportunity to change it during run-time, e.g. to add new network conditions. This evokes the need for reducing the table size.

The reduction process requires the user to specify for each performance metric  $P_i$  if higher or lower values are preferred (e.g. higher delivery ratio and lower energy values are preferred), in which granularities these values should be specified during run-time (e.g. the delivery ratio can only be specified in steps of 5 percentage points), and optionally also the minimal or maximal acceptable value for this metric. Then, the resulting CT will only support goal definitions that respect this specification. The user can only set lower bounds for performance metrics whose values are preferred to be higher (e.g.,  $P_i \geq v$ ), and vice versa. We do not see good reasons to neglect this and, for instance, to specify an upper bound for the delivery ratio to limit the application performance.

Reduction of the CT size is possible for two reasons: First, the CT also contains entries with performance values that are unacceptable for the user. Thus, due to the provision of this information already in the pre-installation phase, bad entries can be removed. Moreover, it can be checked if these minimal requirements can be fulfilled for all scenarios. If this is not the case, the engineer needs to re-configure the application.

Second, even if an entry fulfils the minimal requirements it will not be selected by the adaptation process when another entry is better. This is the case if the other entry is as good or better for all performance metrics  $P_i$  and better in at least one metric. We say that the better entry *dominates* the other.

#### 4 TinyAdapt Implementation

Skyline algorithms known from database systems, e.g. [BKS01], can identify all entries that are dominated by other entries, thus performing the needed reduction. Our implementation uses a naive nested-loop approach which compares every entry with every other entry. Although there are more efficient implementations for database systems the run-time of the algorithm is not the main focus since the reduction process is only run once and it does not have to deal with large amounts of entries.

Let  $X$  be the set of all tuples  $x = (x_1, x_2, \dots, x_p)$  that contain only the performance metrics for each table entry. We denote  $g_i$  ( $g_i > 0$ ) to be the granularity specified by the user and  $m_i = \lceil \frac{\max_{x \in X} x_i}{g_i} \rceil g_i$  be the maximum of  $P_i$  rounded up to the next multiple of the granularity. In the case that lower values are preferred for a performance parameter  $P_i$ , all the values for  $P_i$  are inverted to ease subsequent processing. For these  $P_i$ , set  $x_i = m_i - x_i \forall x \in X$ .

It is important that a valid configuration can be found for all network conditions. Therefore, the following steps are performed separately for each group of table entries that were created by the same exploration sub-scenario. Thus, at least one table entry will remain for each sub-scenario. The possible fusion of entries from different, but similar exploration sub-scenarios is checked afterwards.

The basic idea of the Skyline algorithms is to remove all tuples  $y$  that are dominated by another tuple  $x$  based on the user preference for each  $P_i$ . That is, remove the tuple  $y$  that satisfies the following rule:

$$y \in X \mid \exists x \in X : (y_i \leq x_i \forall i \in [1, p]) \wedge (\exists j \in [1, p] : y_j < x_j) \quad (4.1)$$

The resulting skyline can still be quite large since under certain conditions points that are very close cannot be removed. For example, consider two parameters  $A$  and  $B$  both with higher preferred values. The first entry contains values  $(a_1, b_1)$  for these parameters, the  $n$ -th entry contains values  $(a_{n-1} + \epsilon, b_{n-1} - \epsilon)$  (see Figure 4.7). None of these entries can be removed since no one is *better* than the other. This problem can be mitigated if the users do not use arbitrary values to specify their requirements but use multiples of the granularity  $g_i$ . Then, only the sub-sets  $x \in X : k_i g_i \leq x_i < (k_i + 1)g_i, k_i \in \mathbb{N}$  need to be considered for reduction.

Figure 4.8 illustrates the CT reduction for the mobile sub-scenario of our example application. Each entry of Table 4.2 is represented as a blue cross. Note that for the energy the inversion formula was not applied although smaller values are better. Instead, the y-axis was reversed so that both the reader can better find the table values in the diagram and the skyline algorithm can be applied graphically. The grid is formed by the user-specified granularities, which is 5 for the delivery ratio and 20 for the energy. All data points inside a grid cell including the left and lower border of the cell belong to the same sub-set; the left lower corner of the cell is referred to as the sub-set position.

When removing tuples, now we compare the sub-set position to which they belong. Let  $d_i(z) = \lfloor \frac{z}{g_i} \rfloor, i \in [1, p]$  be the discretisation function for parameter  $i$ . The tuples

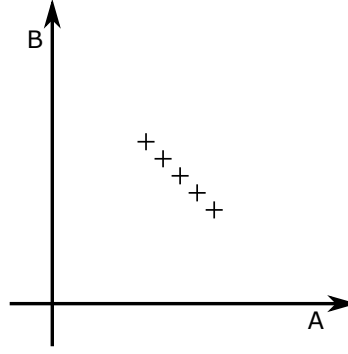


Figure 4.7: Irreducible Configuration Table entries

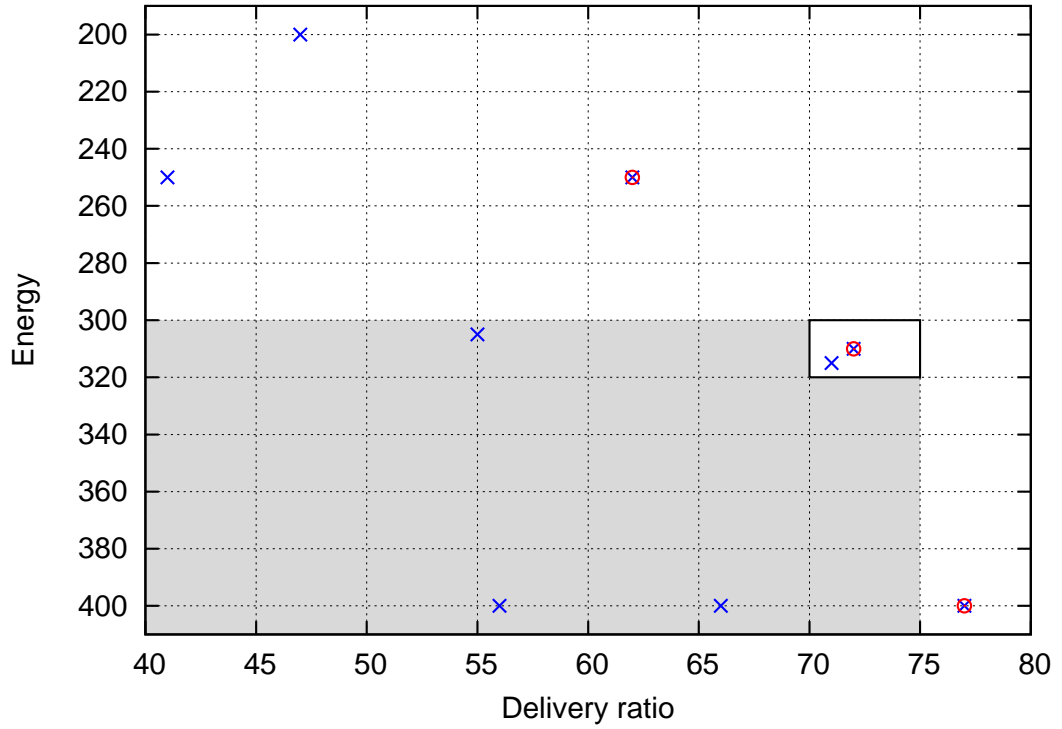


Figure 4.8: Reduction of Configuration Table

are removed as in rule (4.1) but  $d_i()$  is applied to  $x_i$ ,  $x_j$ ,  $y_i$  and  $y_j$  first. This means that the discretisation step is independent from the reduction step and any Skyline algorithm can be used for the latter.

In Figure 4.8 when considering the sub-set in the bold rectangle all data points that are located in the grey shaded region can be removed. As can be seen easily also tuples might be removed that are better by  $g_i - \epsilon$  in all but one parameter  $P_i$ , but by choosing  $g_i$  the user can trade off table size by (partially) better adaptation results.

The remaining sub-sets can contain more than one tuple of which we can select one that represents the sub-set (with a similar trade-off). As default, we sort the tuples of a

## 4 TinyAdapt Implementation

sub-set by the Manhattan distance to the sub-set position, i.e.  $\sum_{i \in [1,p]} |x_i - d_i(x_i)|$ , and take the one with the largest distance. This assumes that all performance parameters are equally important; but other metrics can be included by the engineer. In the example, two data points remain in the bold rectangle after this skyline algorithm has been executed, but the one marked with a red circle has the larger Manhattan distance to the sub-set position and is, therefore, chosen.

In the example, the user has also specified that delivery ratios below 50% are not acceptable. Therefore, these entries can also be removed. Finally, only the three points marked with red circles remain for this sub-scenario. The reduction process is also performed for the static sub-scenario. The resulting CT for the complete scenario is shown in Table 4.3. It contains only 5 entries compared to the original 17 entries before reduction.

$N_1$ Mobility	$A_1$ Beacon Int.	$A_2$ LPL Int.	$P_1$ Deliv. Ratio	$P_2$ Energy
0	60	0	100	395
0	60	250	99	100
1	20	0	77	400
1	20	250	72	310
1	40	250	62	250

Table 4.3: Characteristics Table of example application after reduction

### 4.6.2 Merging of Exploration Results

As described in the previous section the reduction process results in at least one table entry per exploration scenario. Typically, more entries remain in the CT since performance goals are often conflicting. For example, lower power consumption for MAC protocols is often tied with higher latency due to sleep cycles.

However, different exploration scenarios can still result in similar algorithm performance, even with the same parameterisation. For example, an algorithm might be tuned to different node speeds, but above a certain speed it performs similarly. Therefore, it does not make sense to keep the parameterisations for both scenarios. Instead they could be merged.

The merging process compares the performance of all entries of the reduced CT for a scenario A with the performance of the same parameter combinations for another scenario B. Note that these parameter combinations do not necessarily need to be also in the reduced CT set for scenario B. Similar, scenario B is compared to scenario A. If all performance metrics are similar (we use the metric granularity of the reduction process as degree of similarity) the results of both scenarios are merged. In order to decide which of the parameter combinations of the union of A and B are to be kept the reduction process is simply repeated for the union.

In our example application, it is obvious that no scenarios can be merged since in the static scenario the delivery ratio is always above 97% while in the mobile scenario the maximum ratio is 77%.

### 4.6.3 Mapping of Network Metric Values

Different exploration scenarios should result in different network metric values, at least if the algorithms react differently in the scenarios. This is the case for all scenarios that could not be merged in the previous step. To check this, all scenarios are again compared with each other, and for each combination of scenarios the value range of at least one network metric must not overlap.

If two scenarios cannot be distinguished safely by their network metrics the engineer has three possibilities: firstly, the network metrics are refined and the TinyAdapt workflow is repeated starting with step 5 in Figure 3.1 to obtain better separation; secondly, the scenarios are merged anyway resulting in possibly sub-optimal configurations; and thirdly, the results can be kept and adaptation will in some cases select configurations from a different scenario. However, in the latter case results will probably not differ much. TinyAdapt helps the engineer by presenting the conflicting configurations, i.e. the parameter combinations with significantly different performance metrics that prevented merging.

We do not show this here for the example applications, but similar results are presented when evaluating the mobility metric in Section 6.2.

Finally, the scenarios are numbered and this number is stored with the ranges for each performance metric of this scenario in a mapping table. In the CT, the performance metric values are replaced with the scenario number, which saves a lot of space. During run-time, the measured network metric values are compared with the entries in the mapping table using a nearest-neighbour approach to determine the number of the operational scenario. This number will be used to look up suitable entries during the adaptation process.

## 4.7 Adaptation Engine

During run-time, the Adaptation Engine serves as the core of the run-time system: it receives updates of the Goal Definitions from the Distribution component and the current values of the network metrics from the Monitoring component, and after executing the adaptation process it instructs the Installation component to exchange algorithms and sets the parameters of an algorithm using the Settings component.

When a new Goal Definition is received a short delay is applied in order to let the Goal Definition propagate further through the network since the node might be stopped

#### 4 *TinyAdapt Implementation*

during the code exchange process. This propagation delay is determined by the configuration test application (Section 4.2).

The Monitoring component frequently delivers new network metric values, especially in dynamic scenarios. Then, the Adaptation Engine first performs the mapping to the metric classes as described in Section 4.6.3. A change in a metric class needs to be stable for a short amount of time to start the adaptation in order to avoid oscillation or unnecessary adaptation. This stabilisation time is also determined by the configuration test application.

After the propagation delay or the stabilisation time has passed, the adaptation process (see pseudocode in Figure 4.9) is triggered. It is driven by the Goal Definitions. Their structure was already presented in Section 3.4.3. In short, they consist of requirements as well as relaxation and optimisation definitions.

```
while not found
  for each table entry
    if entry matches the network metric mapping
      and entry fulfils all requirements
        mark as candidate
      end if
    end for

    if no candidates were found
      for each relaxation definition
        if constraint can be changed within bounds
          apply relaxation to constraint
          break for
        end if
      end for
      if relaxation was impossible
        stop adaptation
      end if
    end if
  end while

  if more than one candidate
    for each optimisation definition
      find max/min value according to definition
      eliminate candidates that have different value
    end for
  end if
```

Figure 4.9: Adaptation Algorithm

The specified requirements on the performance metrics  $P$  and the mapping of the network metrics  $N$  are used to select suitable entries from the Configuration Table. If no such entries are found the relaxation definitions are applied one at a time, i.e. the boundary of the specified requirement is changed in the specified interval and direction. After each relaxation step the (new) requirements are re-evaluated. If a requirement has reached the boundary defined by a relaxation rule, the next relaxation rule is applied using the same process. If there is no next rule, the adaptation stops since it is not possible to find a matching table entry. If finally at least one matching table entry is found, the entry that matches best the optimisation rule will be selected, i.e. the entry that includes the maximum or minimum value for the specified table column.

Requirements	1. delivery ratio $\geq 90\%$ 2. energy $\leq 200J$
Relaxation	1. decrease requirement 1 in steps of 10 until 70 2. increase requirement 2 in steps of 100 until 400 3. decrease requirement 1 in steps of 10 until 50
Optimisation	minimise energy

Table 4.4: Example Goal Definition

We will now apply the Goal Definition of Table 4.4 to the Characteristics Table of our example application (Table 4.3). In the mobility class 0, only the second entry immediately matches the requirements (see Table 4.5) and the adaptation process ends.

$N_1$	$A_1$	$A_2$	$P_1$	$P_2$
Mobility	Beacon Int.	LPL Int.	Deliv. Ratio	Energy
0	60	0	100	395
0	60	250	99	100

Table 4.5: Adaptation process for static example application

In the mobile case, no entry is found that matches the requirements. Therefore, the relaxation rules are applied (see Table 4.6). First, the requirement on the delivery ratio is lowered until 70%, but still no table entry can be found. Since relaxation rule 1 does not allow further relaxation the second rule is applied. After raising the available energy two times to 400, table entries match.

In fact, two table entries now fulfil the (relaxed) requirements (see Table 4.7). Therefore, the optimisation rule is applied and the second entry is finally selected.

Since all nodes operate on the same goal definition and have the same network metrics values they will select the same entry using this algorithm. Therefore, no further agreement step is necessary. If a new algorithm is selected it is installed using the Installation component. New algorithm parameters  $A$  are set using TinyXXL, which,

## 4 TinyAdapt Implementation

Relaxation		Requirement for	
Round	Rule	Del. Ratio	Energy
(original)		$\geq 90$	$\leq 200$
1	1	$\geq 80$	$\leq 200$
2	1	$\geq 70$	$\leq 200$
3	2	$\geq 70$	$\leq 300$
4	2	$\geq 70$	$\leq 400$

Table 4.6: Application of the relaxation rules in example

$N_1$	$A_1$	$A_2$	$P_1$	$P_2$
Mobility	Beacon Int.	LPL Int.	Deliv. Ratio	Energy
1	20	0	77	400
1	20	250	72	310
1	40	250	62	250

Table 4.7: Adaptation process for mobile example application

in turn, notifies the parameterisable algorithms that one of its parameters has changed. The algorithm is then responsible for reading the new value from TinyXXL and acting accordingly.

If the installation involves a node restart or reinitialisation the current state of the Adaptation Engine needs to be stored to non-volatile memory. For wireless sensor nodes, the EEPROM of the CPU is used here. This state includes the current Goal Definition and the current network metric values so that the adaptation process would result in the same result as before the restart.

Each time the adaptation process is invoked it starts with the original Goal Definition, i.e. the “unrelaxed” requirements. Therefore, all relaxation operations, which change the requirements, have to be performed on a copy of the Goal Definitions only. This extra memory cannot be avoided. However, memory can be saved elsewhere, e.g. by using bitfields to mark all candidates.

## 4.8 Conclusion

A realisation of the theoretical adaptation framework developed in Chapter 3 has been presented in this chapter for TinyOS. We have shown a new monitoring component that calculates and distributes extreme values of network metrics in an efficient way. A novel mobility metric delivers more stable measurements of mobility to drive adaptation. The problem of algorithm parameter space exploration, the filtering of relevant results and storing them on the sensor nodes was solved as well. Finally, the actual adaptation



process was presented in detail. By executing each step with an example application, we showed the feasibility of the TinyAdapt approach.

In fact, TinyAdapt copes with the challenges that were only partly solved by existing adaptation solutions: TinyAdapt is a generic framework that works with arbitrary algorithms, algorithm parameters and network metrics; it provides support to find the best algorithms and parameters for a certain scenario; during run-time it executes a distributed and dynamic adaptation process driven by an explicit and easy definition of adaptation goals. In Chapter 6 run-time results for TinyAdapt will be presented, showing that adaptive applications achieve significant performance improvements compared to their non-adaptive counterparts.



# 5

## **TinySwitch - Switchable Components for TinyOS**

Besides parameterisation of algorithms, TinyAdapt supports the complete exchange of an algorithm as a mean of adaptation. Section 2.3.2 reviewed existing code replacement solutions for TinyOS. Many of these solutions reprogram the flash ROM of the microcontroller, which consumes a considerable amount of power (see Section 6.7.3) and may fail since a minimal voltage of 2.7V is required for both MicaZ and TelosB motes. Moreover, the application is totally stalled during reprogramming, which can take several 100ms. Some TinyOS approaches even require a node restart, including a complete node reinitialisation. Other solutions that avoid reboot fail in proper initialisation of the new code.

To increase adaptation efficiency by avoiding the flash reprogramming and node reinitialisation we propose to include all alternative algorithms for a task in one single binary image, while ensuring that exactly one selected algorithm per task is active at a time and all others remain inactive. Our solution, TinySwitch, also ensures that all non-active algorithms do not interfere with the rest of the application. Existing solutions that propose algorithm switching and have also been described in Section 2.3.2 either do not provide a general solution for arbitrary algorithms or require considerable work by the user, such as providing component models or manual switching components. The TinySwitch framework supports the programmer in creating a switchable application by analysing the interfaces of the alternative implementations and by generating code to implement aforementioned switching features. It is worth noting that our solution does not exclude reprogramming but can be viewed as an orthogonal approach that can co-exist with the reprogramming techniques, e.g. to replace faulty code.

### **5.1 Generic Switching Concept**

This section presents the general idea and architecture of algorithm switching, requirements to the switchable algorithms concerning their interfaces, and optimisation possibilities to improve application performance.

### 5.1.1 Switching Architecture

Since code should not be replaced by modifying the running program in order to avoid flash reprogramming, all alternative algorithms already need to be included in the installed binary. As consequence, the control flow of the application is altered logically so that only the wanted algorithm is executed. Figure 5.1 shows the general structure of a software architecture with switchable algorithms.

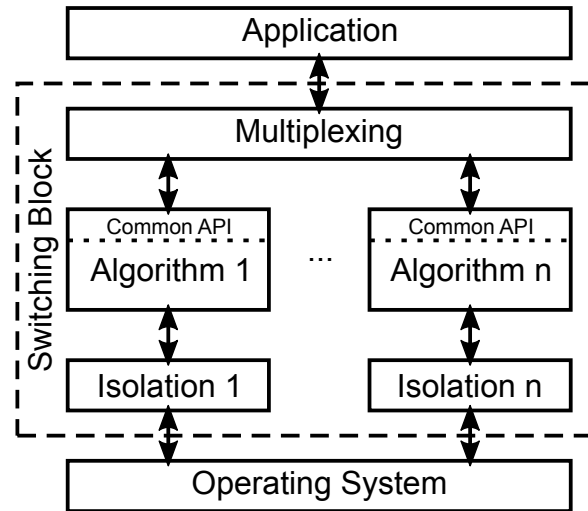


Figure 5.1: General switching architecture

All interchangeable algorithms, i.e. all algorithms of the same algorithm class that could be alternatively used for the same task such as routing or time synchronisation, form a **switchable algorithm set** and for each set a **switching block** is generated. Multiple switching blocks can coexist in one application, both horizontally, e.g. for time synchronisation and localisation, and vertically, e.g. for routing and MAC, in a layered architecture.

At any time, exactly one algorithm of a switching block is enabled. The application calls functions of the **multiplexing layer** that forwards the calls to the enabled algorithm. The multiplexing layer also forwards calls only from the enabled algorithm to the application. Between the algorithms and the operating system, the **isolation layer** lets pass calls only from and to the enabled algorithm. If a call cannot be forwarded since the algorithm is disabled a default value specified by the programmer is returned. This architecture assumes that the operating system already calls the correct algorithm, e.g. because they are bound to different resources. If this is not the case, the isolation layer also needs to provide some multiplexing functionality.

It is the primary goal of TinySwitch to generate these multiplexing and isolation layers. Applying the design principles established for TinyAdapt in Section 3.2, this goal should be achieved in a mostly automatic fashion and with minimal extra manual work for the user (*ease of use* principle) and switching should be possible for arbitrary

algorithm classes (*generic applicability* principle). Moreover, the solution should be light-weight during run-time.

### 5.1.2 Algorithm Requirements

To automatically provide the multiplexing layer and the isolation layer, the switchable algorithm must comply with a few requirements concerning its interfaces towards adjacent layers. For the sake of convenience, we denote the layer above the switchable algorithm as **application** and the layer below it as **operating system**. However, this does not restrict the applicability of our solution since the top layer can also be another algorithm, e.g. a data management algorithm if we create switchable routing algorithms, and the bottom layer can be another algorithm in combination with the operating system, e.g. a switchable routing algorithm calls both the underlying MAC algorithm (for sending packets) and the operating system (for timers etc.). The switchable algorithm (in the following only **algorithm**) provides a well-defined application programming interface (**API**) through which it is accessed.

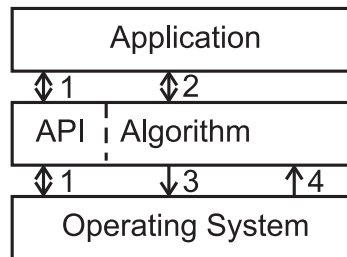


Figure 5.2: Different calls to and from algorithm

Figure 5.2 shows possible calls to and from an algorithm. In order to be suitable for the switching architecture, some restrictions apply to these calls. The application or the operating system must only call functions in the API of the algorithm (Link 1) and callbacks from the algorithm are also allowed only through the API (indicated as double sided arrow). Calls to any other algorithm function are disallowed (Link 2) as well as calls from the algorithm to the application bypassing the API. On the other hand, the algorithm can call functions in the operating system (Link 3) to perform its operations. In general, it cannot be safely detected if the call from the algorithm goes to the application or the operating system. However, the former will hardly happen if general-purpose algorithm implementations are used, which should not depend on application features that are not in the API. In contrast, calls from the algorithm to the operating system also create dependencies, but the functions of the operating system can be assumed to exist always. Calls from the operating system to the algorithm (Link 4) can be either callbacks for notifications, which are allowed, or to request algorithm operations, which are disallowed. In the first case, the callback was registered before by the algorithm and the notification can simply be ignored if the algorithm is disabled without unregistering the callback. Without the

callback registration the operating system would not even know about the algorithm and, thus, also has no static reference to it. In the second case, it can be assumed that the operating system depends on the algorithm and, thus, the algorithm cannot be disabled at all. In contrast to the first case, the operating system would always have a reference to the algorithm, which can be used to distinguish both cases.

In summary, only algorithms that adhere to these rules can be switched because they have a well-defined API and only minimal dependencies. To automatically provide the multiplexing layer and the isolation layer, all calls to and from the algorithm need to be analysed. This process heavily depends on the programming language and such analysis can be performed on source code, inside or after a possible precompiler run, inside the compiler or using the final binary. Due to heavy compiler optimisations the latter is often the most difficult variant.

When algorithms are switched it is advisable to explicitly stop the old algorithm and to start the new algorithm in order to let them cleanly stop internal operations and initialise itself, respectively. For this purpose, the algorithms need to provide explicit start and stop functions in the API.

The algorithm to be stopped could also store and transfer its state to the new algorithm if both algorithms support this. For example, in TinyOS, they could use a shared data repository like Chi [FET<sup>+</sup>10] or TinyXXL [LMM<sup>+</sup>06] to store state so that all algorithms can use it if needed. Note that this is not specific to algorithm switching but can also be applied when replacing code.

### 5.1.3 Defining Common APIs

All algorithms in a switching block are accessed through the interface of the multiplexing layer (see Figure 5.1). If algorithm 1 had a different interface than algorithm 2 the multiplexing layer would need to handle the calls to its interface in a different way when forwarding them to algorithm 1 than when forwarding to algorithm 2. As consequence, the automatic generation of the multiplexing layer would not be feasible anymore. Therefore, the switching concept requires all interchangeable algorithms to expose the same interface.

This API unification is a common approach for the creation of interchangeable components. If two components have the same interface a program can be linked with either of them and there is no need to change, for example, function calls. As shown in Section 2.3.2, also existing TinyOS solutions that exchange parts of the code (TinyModules, ELON) require the programmer to define a common interface.

In our switching concept, the common API has to contain all functions of the algorithms that are called by the application or the operating system and all callbacks from the algorithm to the application (Link 1 in Figure 5.2). There is no need to also include functions on the operating system that are called by the algorithms (Link 3 in Figure 5.2) since these links can be detected and automatically disconnected in

the isolation layer when an algorithm is deactivated. This is an advantage over other TinyOS solutions where these functions also need to be included in the interface.

The functionality of the common API is usually the minimal common set of all algorithm APIs. It has to be defined by the programmer using knowledge about the application and the single algorithms. For all algorithms with a native API that is different from the common API, wrappers have to be built to translate the calls. Typically, default parameters have to be added or a single call has to be translated into calls to more than one function. For example, a tree-based routing algorithm has a function designating a node as root and one for sending a packet to the root node without the need to specify a destination address. In contrast, a simple flooding routing algorithm only has a function for sending a packet to an arbitrary address. The programmer defines the common API to contain an “init” and a “send” function. In the wrapper of the tree-based routing algorithm, the “init” function calls the function to make a node X root; in the wrapper of the flooding algorithm it is empty. The “send” function of the wrapper of the tree-based routing algorithm simply forwards the call to the native interface, while the wrapper for the flooding algorithm sets the address of node X as default destination for each packet.

### 5.1.4 Manual Optimisations

Using the general switching architecture described before all calls to/from the algorithms could be forwarded or rejected depending on whether the algorithm is enabled or disabled. This straightforward solution results in a working application that can already be used. In the following, we discuss three situations where the performance of an application using algorithm switching can be optimised when more information on the type of calls is available. In some cases, this also requires manual code inspection or changes.

First, an algorithm could start external operations that would continue even if the algorithm is disabled. For example, a periodic timer could have been started when the algorithm was active, and although the timer event will not reach the disabled algorithm it still consumes resources in the operating system and could shorten the sleeping times of a node. Therefore, if the analysis process knows the functions dealing with the timer it can check if any of these is called by the algorithm and ask the programmer during analysis to check if the timers are properly stopped when the algorithm is switched off.

For example in TinyOS, an algorithm uses the `Timer` interface and starts a periodic timer at some point:

```
module DataRouter {
  uses interface Timer;
  ...
}
```

```
implementation {
    ...
    call Timer.startPeriodic(10240);
    ...
}
```

The programmer has to ensure that stopping the algorithm also stops this timer:

```
command error_t StdControl.stop() {
    ...
    call Timer.stop();
    ...
}
```

Second, stopping an algorithm and restarting it minutes later can also lead to undesirable effects. For example, if packets are already in the send queue of a routing algorithm that is stopped, these packets should not be sent when the algorithm is restarted again later since they are most likely stale. Rather, the queue should be emptied if the state cannot be transferred to the new algorithm. If the algorithm manages its queue by external functions and the analysis process knows them it can notify the programmer during analysis to check for the necessity of such a manual change.

For example in TinyOS, a routing algorithm could use the `Queue` interface to realise such a sending queue:

```
module DataRouter {
    uses interface Queue<message_t*>;
    ...
}
implementation {
    command error_t Send.send(message_t *msg, uint8_t len) {
        ...
        call Queue.enqueue(msg);
        ...
    }
}
```

The programmer could add the following code for a cleaner stop:

```
command error_t StdControl.stop() {
    ...
    while (! call Queue.empty()) {
        msg = call Queue.dequeue();
        // do something here with msg, e.g. notify the
        // application that msg could not have been sent
    }
    ...
}
```



Finally, if a function call does not change the state of the callee or if such state change is negligible, this function does not need to be isolated at all. Examples are getter functions that only return values or setter functions that set values of a structure given as parameter. If the analysis process knows these functions they will not be included in the isolation layer. However, such functions need to be multiplexed since they might have different behaviour in different algorithms.

In TinyOS, a typical example would be the `AMPacket` interface. There, all `set*` commands accept a message pointer as parameter, and the corresponding field is only changed in this message:

```
interface AMPacket {
    ...
    command am_addr_t destination(message_t* msg);

    command void setDestination(message_t* msg, am_addr_t addr);
    ...
}
```

As only `msg` is changed there is no need to multiplex or isolate this interface.

## 5.2 TinySwitch Operation

Based on the concept of the aforementioned algorithm switching architecture, this section presents TinySwitch to support run-time switching for TinyOS applications. The use of TinySwitch consists of several steps. First, a common API for all algorithms in a switchable algorithm set needs to be created. Afterwards, a test application has to be built with each of these algorithms. TinySwitch analyses these applications and determines the interfaces and functions that the multiplexing and the isolation layer need to take care of. For known interfaces, TinySwitch gives hints to the programmer for possible optimisations. If the programmer then decides to change the code, the analysis step must be repeated. When all algorithms of a switchable set have been analysed, TinySwitch can generate the code for the multiplexing and isolation layer. This usually happens later in step 9 of TinyAdapt’s workflow (see Figure 3.1) when the final application is being built with TinyAdapt. However, TinySwitch work also as standalone switching framework, and the multiplexer’s interface can be used directly by the application to switch algorithms.

During the detailed presentation of these steps in the following sections we apply them to a sample application that includes two routing algorithms: TinyOS’ Collection Tree Protocol (CTP) [FGJ<sup>+</sup>07] and a typical Flooding algorithm. CTP is a routing protocol that is used for transmitting data from one or multiple sources to one or multiple roots following a tree-structured routing path, while the Flooding algorithm delivers the packets by “flooding” them to the whole network. As we see later on in the

Evaluation chapter (Section 6.6) CTP works very well in static and stable scenarios while Flooding can be used in mobile or other unstable scenarios.

### 5.2.1 API Unification

As explained in Section 5.1.3, the algorithms of a switchable set need to have the same API. In TinyOS, this means that the components representing the API need to provide and use the same interfaces. Then, an algorithm can be replaced by another by just replacing the “components” statements in the configuration’s wiring. For example, a simple monitoring application could include a routing algorithm in its configuration file as follows:

```
configuration SimpleMonC {}  
implementation {  
    ...  
    components SimpleMonM;  
    components new RoutingAlgoC(123) as RoutingC;  
    SimpleMonM.Send -> RoutingC;  
    ...  
}
```

If another routing algorithm provides the same interfaces, both algorithms can be exchanged by replacing the second “components” line with:

```
components new OtherRoutingAlgoC(123) as RoutingC;
```

No other changes are necessary, in particular all the wiring in this configuration remains the same.

Beside the translation of calls to be implemented in the wrappers as explained in Section 5.1.3 some features of the nesC language require special attention in the definition of the common API. These are explained in the following.

### Generic Components and Parameterised Interfaces

Components can use or provide so-called parameterised interfaces, e.g. `uses interface X[uint8_t id]`, which corresponds to multiple interfaces of type X that can be distinguished by the integral parameter `id`. If a second algorithm uses or provides X without these instance parameters, the wrapper configuration of the first algorithm usually declares only a single interface and wires it to a constant `id`:

```
// algorithm A  
uses interface X[uint8_t id];  
  
// algorithm B
```

```

interface X;

// wrapper configuration for A
configuration WrapperA {
    uses interface X;
}
implementation {
    components A;
    X = A.X[123];
}

```

A similar approach is feasible for so-called generic components that can be instantiated multiple times (non-generic components share a single instance for the whole application) and can be parameterised with component parameters. For example, **generic module** `A(typedef t)` accepts a C type as parameter that can be used inside the module for declarations. If the API component of the first algorithm is generic and the API component of the second algorithm is not, the wrapper configuration of the first algorithm is often non-generic and instantiates the original API component with a constant parameter:

```

// algorithm A
generic module A(typedef t) {
    provides interface GetNow<t>;
    ...

// algorithm B
module B {
    provides interface GetNow<bool>;
    ...

// wrapper configuration for A
configuration WrapperA {
    provides interface GetNow<bool>;
}
implementation {
    components new A(bool);
    GetNow = A.GetNow;
}

```

A third wrapper configuration is of interest especially for routing algorithms, where, for example, algorithm A contains a generic API component with non-type parameters and inside a non-parameterised interface (I) and algorithm B contains a non-generic API component and inside a parameterised interface. Here, the wrapper configuration for algorithm B can be generic and the wiring translates the generic component to a parameterised interface:

```
// algorithm A
generic configuration A(uint8_t id) {
    provides interface I;
    ...

// algorithm B
configuration B {
    provides interface I[uint8_t id];
    ...

// wrapper configuration for B
generic configuration WrapperB(uint8_t id) {
    provides interface I;
}
implementation {
    components B;
    I = B.I[id];
}
```

### Control Interfaces

In *TinyOS*, two different interfaces allow for starting and stopping of algorithms. **StdControl** is commonly used if the algorithm can be switched on and off instantaneously. This is usually the case if the algorithm needs not to switch on/off hardware. In contrast, with **SplitControl** the **start** command can power on needed hardware in turn. Later, the hardware signals its readiness to the algorithm with a **startDone** event, which in turn is usually signalled to the user of the algorithm.

With **StdControl**, switching of algorithms can be done in an atomic way. In contrast, with **SplitControl** control needs to be returned to the operating system after calling **start** or **stop** of the algorithm, and the switching process can continue only after receiving a **startDone** or **stopDone** event. Thus, there is a short time period in which no algorithm is running.

For this reason, the use of **SplitControl** should be avoided. The most efficient way is to change the algorithm in such a way that it already provides **StdControl** instead of **SplitControl**. Since algorithms of the same type also often use the same hardware this approach is beneficial. For example, usually all routing algorithms need a working radio and, therefore, the radio startup can be moved from the algorithm to the application and the interfaces can be changed to **StdControl**. Nevertheless, some algorithm components might depend on **startDone** and **stopDone** events from the hardware to work properly. To ease the modification of the algorithm, *TinySwitch* can fake these events, which is explained in Section 5.2.2.

If such modification is impossible for all algorithms of a switchable set, e.g. because some hardware is used by only one algorithm, and not all algorithms already provide the same control interface, wrappers need to be created for interface unification. It is easy to convert a `StdControl` interface to a `SplitControl` interface in the wrapper: the `SplitControl.start` command calls `StdControl.start` and posts a task that later on signals `SplitControl.startDone`. The same procedure can be applied to the `stop` command. Now, all the algorithms suffer from the split phase switching problem described above.

The effort for a conversion of `SplitControl` interface to a `StdControl` interface depends on how the algorithm handles calls to other interfaces than `SplitControl` when the hardware has not been started. If the algorithm keeps track of the status of the hardware and either returns an error code or defers the execution of the command, e.g. by putting a packet in a send queue, no additional logic is required in the unification components. However, if the algorithm relies on the correct call order it has to be protected with additional wrapper code from wrong invocations. The `StdControl.start` command calls `SplitControl.start` and the resulting event `SplitControl.startDone` sets an internal flag that shows that startup has been completed. All calls to other interfaces of the algorithm first need to check this flag and return an error (`EOFF`) if the flag is not set. Otherwise, the call can be forwarded.

### **Example: Unification of CTP and Flooding**

Our sample application shall switch between CTP and Flooding. Therefore, their APIs need to be unified. After inspecting the CTP and Flooding APIs, we can highlight two main differences. First, CTP additionally provides the `RootControl` interface that is used by the application to specify the root of the collection tree. Second, since all packets are transmitted to the root, CTP provides the `Send` interface to send the packets without specifying the destination. In contrast, Flooding provides the `AMSend` interface, which expects a packet destination. A common denominator could be to send all packets to a fixed data sink, which is the root node in CTP and a static destination address in Flooding. Then, the `Send` interface can be used as the common interface.

The interface unification process involves (1) defining the unified APIs for both CTP and Flooding, (2) converting the `AMSend` interface used by Flooding into the `Send` interface and (3) to hide the `RootControl` interface. Figure 5.3 illustrates the resulting unification APIs for both routing algorithms.

The three components at the bottom (`CollectionC`, `CollectionSenderC`, `FloodingC`) are the original components used to start/stop the algorithms and to send/receive packets. First, to define the unified APIs, for each algorithm we provide a configuration `RtXXXCtrlC` and a generic configuration `RtXXXSendRecvC`, where `XXX` refers to the name of the routing algorithms. Both `RtCtpSendRecvC` and `RtFloodSendRecvC`

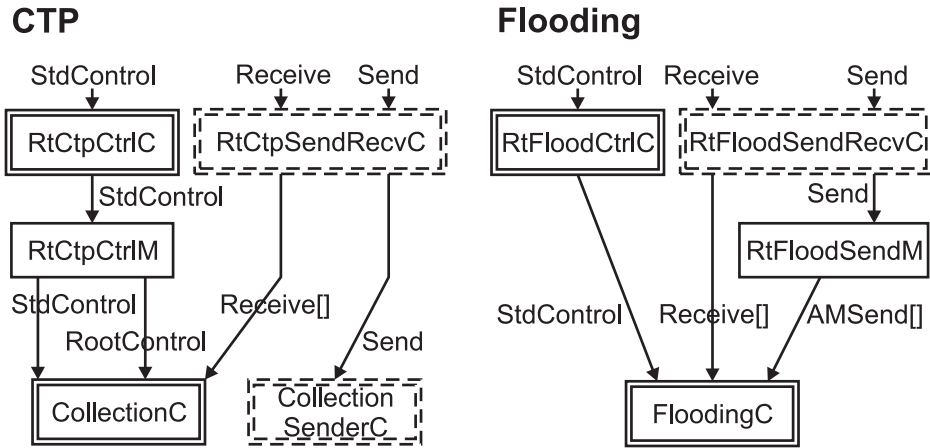


Figure 5.3: Unification of interfaces for CTP and Flooding

are parameterised with the AM id, which is used to wire the **Receive** and **Send** interfaces to their parameterised versions (e.g. **Receive[]**) or to create instances of other parameterised configurations (e.g. **CollectionSenderC**):

```
// RtCtpSendRecvC
generic configuration RtCtpSendRecvC (uint8_t am_id) {
  provides {
    interface Send;
    interface Receive;
  }
}

implementation {
  components new CollectionSenderC(am_id);
  Send = CollectionSenderC.Send;

  components CollectionC;
  Receive = CollectionC.Receive[am_id];
}

// RtFloodSendRecvC
generic configuration RtFloodSendRecvC (uint8_t am_id) {
  provides {
    interface Send;
    interface Receive;
  }
}

implementation {
  components FloodingC;
  components RtFloodSendM;
```

```

Send = RtFloodSendM;
RtFloodSendM.AMSend -> FloodingC.AMSend[am_id];
Receive = FloodingC.Receive[am_id];
}

```

Second, for Flooding, the `RtFloodSendM` module is added to translate a `Send.send` call to an `AMSend.send` one by setting the default data sink's address as destination address for every packet:

```

// RtFloodSendM
module RtFloodSendM {
  provides interface Send;
  uses interface AMSend;
}
implementation {
  command error_t Send.send(message_t* msg, uint8_t len) {
    return call AMSend.send(SINKNODE, msg, len);
  }

  ...
  // all other (AM)Send command/events are just forwarded to
  // the corresponding command/event of the second interface
}

```

Finally, for CTP, we added the `RtCtpCtrlM` module, which declares the default data sink to be the root of the collection tree when the algorithm is started. This module is wired from the `RtCtpCtrlC` configuration:

```

// RtCtpCtrlC
configuration RtCtpCtrlC {
  provides {
    interface StdControl;
  }
}
implementation {
  components CollectionC;
  components RtCtpCtrlM;

  StdControl = RtCtpCtrlM;
  RtCtpCtrlM.RoutingControl -> CollectionC;
  RtCtpCtrlM.RootControl -> CollectionC;
}

// RtCtpCtrlM
module RtCtpCtrlM {
  provides interface StdControl;
}

```

```

    uses interface StdControl as RoutingControl;
    uses interface RootControl;
}
implementation {
    command error_t StdControl.start() {
        error_t error = call RoutingControl.start();
        if (error == SUCCESS) {
            if (TOS_NODE_ID == SINKNODE) {
                call RootControl.setRoot();
            }
        }
        return error;
    }

    command error_t StdControl.stop() {
        return call RoutingControl.stop();
    }
}

```

Finally, the `RtFloodCtrlC` just passes the control interface of `FloodingC` through, i.e. it separates `StdControl` from the rest of the interfaces that are provided by `FloodingC` to match the common API:

```

// RtFloodCtrlC
configuration RtFloodCtrlC {
    provides {
        interface StdControl;
    }
}
implementation {
    components FloodingC;

    StdControl = FloodingC;
}

```

With these simple modifications we are able to easily exchange CTP by Flooding and vice versa during compile time by changing the “components” statement as stated at the beginning of this section.

## 5.2.2 Wiring Analysis

As shown in Section 5.1.2, the dependencies between an algorithm and its external components need to be analysed to control the generation of the multiplexing and isolation layers. Since in TinyOS direct function invocations to other components are



achieved by calling commands and signalling events in explicitly specified interfaces it is sufficient to analyse the wiring between components, which greatly reduces complexity. Bare commands and events are also possible as connections, but they also need to be specified explicitly in the configurations.

For analysis, the programmer builds a separate application with each algorithm of a switchable set, which is easy if all algorithms have a common API as described in Section 5.2.1. When the nesC compiler is invoked with the “**wiring**” option, the file `wiring-check.xml` in the `build` directory will contain information about all components, interfaces and their wiring<sup>1</sup> of the application after successful compilation. This XML file is the main source of information for TinySwitch’s analysis, which is easier than analysing pure source code or even the binary.

### Wiring Requirements

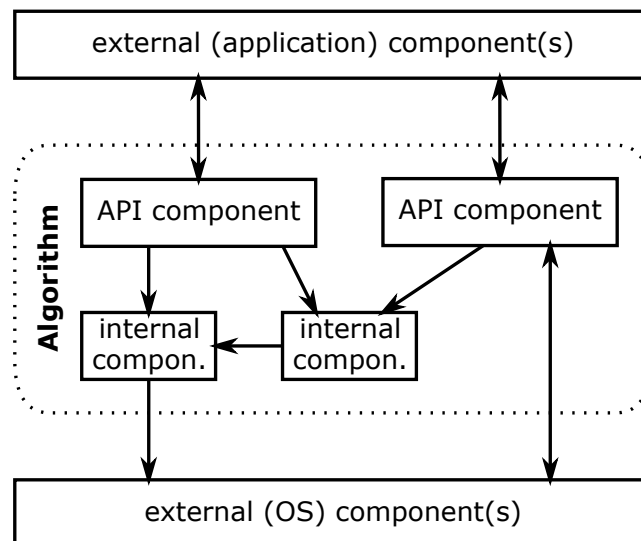


Figure 5.4: Algorithm, components and connections

Figure 5.4 illustrates important terminology for the analysis step. We call the algorithm components to which the application connects **API components** and all other algorithm components **internal components**. Components that do not belong to the algorithm are called **external components**. As explained in Section 2.1.4 the configuration that contains the wiring to establish a connection is usually not shown in these graphs. However, they are also either part of the algorithm, i.e. internal/API components, or not (external component). This distinction is important for TinySwitch as will be explained below.

<sup>1</sup>If the algorithm provides or uses bare functions, `-fnesc-dump=functions` needs to be added to `WIRING_CHECK_FLAGS` to include also the bare functions.

TinySwitch parses the wiring-check.xml file and classifies each component as external, internal or belonging to the algorithm's API. For this purpose, the programmer provides a list of directory paths and files that contain the API components and internal components of the algorithm. For well structured code and especially for algorithms in the TinyOS library this is straightforward. Then, TinySwitch checks all connections and finds those of which the three involved components (providing component, using component, wiring configuration) are not only external or not only algorithm components.

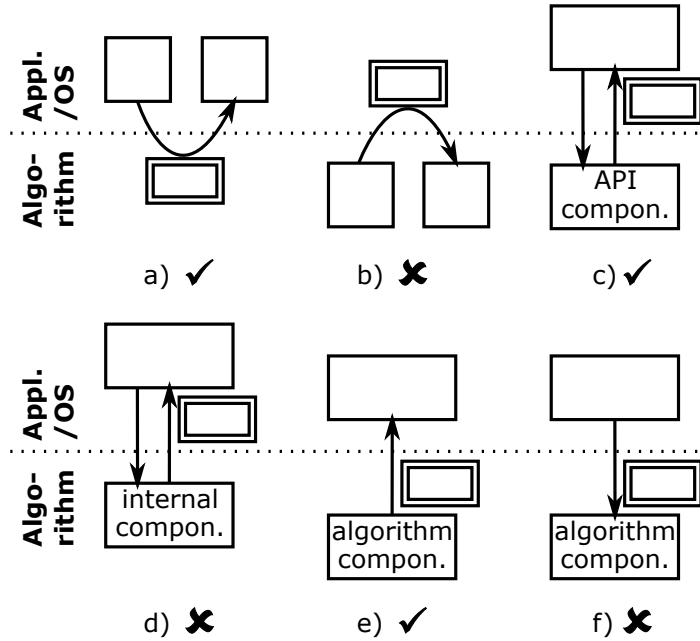


Figure 5.5: Supported connections between external and algorithm components

Each connection that involves both external and algorithm components needs to be examined by TinySwitch to determine if at all and how it should be treated in the multiplexing and isolation layer. For this purpose, we have mapped the allowed and disallowed function calls of Figure 5.2 to TinyOS wirings. Figure 5.5 shows all possibilities how two components (single-border rectangle) can be connected by a configuration (double-border rectangle) with one or two of them belonging to the algorithm. The component providing an interface or bare function is marked by the arrow head. The checkmark or cross indicates if this wiring is allowed and can be handled by the framework.

In wirings a) and b), two external or two application components are connected, which is not a problem at first sight. However, the configuration where this wiring takes place is internal or external, respectively, and thus the wiring also needs to be changed when the algorithm is exchanged. Such a wiring can have different reasons: Firstly, the list of the directory paths or files containing the algorithm components could be incomplete and, thus, the configuration is simply misclassified. The analysis should be re-run with an updated path/file list. Secondly, it could also be caused by bad software design.

As solution for b), the wiring is moved to the wrapper component (see Section 5.2.1), which belongs to the algorithm, to make the connection completely internal. Thirdly, concerning wiring a), the figure might not show the complete picture. Figure 5.6 shows a combination of wirings a) and e): algorithm component D uses an interface in external component B, which in turn uses an interface in external component A, and all wirings are done in algorithm configuration C. This happens when B is a general purpose component (usually generic) that needs another helper component such as a timer. It is B's job to disable A if B is being disabled. Therefore, TinySwitch only cares about the connection between B and D, i.e. wiring e) in Figure 5.5, and prints out notifications about the connection between A and B, i.e. wiring a) in Figure 5.5, that the programmer should check component B on how it manages A.

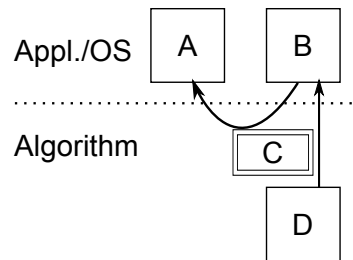


Figure 5.6: Algorithm component D depends transitively on external component A

Wirings c) and d) correspond to the function calls (1) and (2) in Figure 5.2 and are allowed and forbidden, respectively, since the algorithm must be wired from external components only via the API. TinySwitch also checks if the wiring affects one of the control interfaces `StdControl` or `SplitControl`. If so, it needs to be the only control interface in the API to avoid ambiguities how to start and stop the algorithm during switching<sup>2</sup>. Additionally, the component providing the control interface must not be generic since there needs to be a single instance of `StdControl`/`SplitControl` to switch on/off all instances of the algorithm.

Since a user of an interface can call commands in the providing component and the providing component can signal events to the user, allowed wiring e) matches function calls (3) in Figure 5.2 and notification callbacks, which are part of (4). The function calls that cause actions in (4) correspond to wiring f) where an external component uses an interface provided by an internal component, which is forbidden. Since a TinyOS configuration can not only wire interfaces of components that are instantiated by this configuration but can also provide or use interfaces that are provided or used by an internally instantiated component (“wired through interfaces”) a combination of e) and f) can actually be a). Therefore, the analysis follows the connections inside an algorithm if they are just passed through a configuration until an endpoint in an instance of an external or an algorithm component is reached to distinguish between the three possibilities and to treat them accordingly.

<sup>2</sup>An alternative would be that the programmer can specify which one is the main control interface.

### Special Wiring Considerations

Wirings of type e) need special attention in some cases. First, TinySwitch requires the programmer to specify how to handle the interfaces involved in these wirings. The default action is to simply isolate the interface (action *rewire*). For non-void functions, the programmer only needs to specify default values that are returned to the caller by the isolation layer when the algorithm is switched off. Additionally, the optimisations described in Section 5.1.4 can be applied here: action *manual* prints out a message requesting manual checks by the programmer; action *notify* gives hints for optional optimisations; and action *pass* just ignores the interface and connects it through in the isolation layer. TinySwitch already provides reasonable actions and default return values for a range of commonly used TinyOS interfaces.

Algorithms can post TinyOS tasks, e.g. to decouple events from actual processing. The nesC compiler translates posting and executing tasks to calling `postTask` and signalling `runTask` of the `TaskBasic` interface, resulting in wiring of type e). Since TinySwitch does not modify the nesC compiler but works similar to a precompiler it cannot change this wiring automatically. It is also not possible for the programmer to unpost tasks when the algorithm is stopped since this is not supported by TinyOS. Therefore, TinySwitch asks the programmer to modify the tasks to check if the algorithm is still enabled before execution.

Algorithms can also use `StdControl` or `SplitControl` to power on and off external components. As explained in Section 5.2.1 the power control of these external components can be often moved outside of the switchable algorithms in order to speed up switching. To ease the modification of algorithms, TinySwitch supports one special action for `StdControl` and three for `SplitControl`. The *cut* action does simply not forward the `start` and `stop` command of `StdControl` to external sub-components but returns `SUCCESS` or `FAIL` depending on whether the algorithm is enabled or disabled, respectively. Similar, the *call* action answers both commands of the `SplitControl` interface. Additionally, if the algorithm is enabled it signals the corresponding `startDone` or `stopDone` event with a `SUCCESS` argument. Finally, an algorithm might only react to the `xDone` events without actually calling `start` or `stop` itself but relying on external components (typically the application) calling them. In this case, the *outside* and *inside* actions make TinySwitch fake the `startDone` event before or after starting the algorithm, respectively, and to fake the `stopDone` event after or before stopping the algorithm, respectively. Manual inspection of the algorithm's code is necessary to detect which special handling is required, but this is usually an easy task since start and stop sequences are well defined in TinyOS. Of course, the *rewire* action can be used for `StdControl` or `SplitControl` if the algorithm should be able to start and stop the underlying components using one of these interfaces.

The `Init` interface also requires special handling since it is the only exception where wiring e) is forbidden and f) is allowed. A component can provide this interface, and the command `Init.init` is called during TinyOS' boot sequence to initialise the component. This command should not start any services (this is the task of the control

interfaces) and, thus, it can also be executed for disabled algorithms in order to start them later. TinySwitch will not multiplex the `Init` interface and, thus, the interface must not be part of the algorithm's API but only be provided by internal components. In contrast, an internal component must not call `init` in an external component since this might result in multiple initialisations of the external component if that component is used several times. Another point needs to be mentioned: Typically, algorithms are not designed to be stopped and started again after a long time, which would require re-initialisation. Therefore, the analysis prints a note suggesting that parts of the `init` code may be moved or copied to the `start` or `stop` command of the algorithm's control interface.

## Analysis Results

As described, TinySwitch analyses the connections in an application and guides the programmer how to ensure and optimise correct switching behaviour for certain interfaces. If the programmer changes the code, the application needs to be re-compiled and the analysis to be re-run.

The wiring analysis results in an XML file for each algorithm containing information about all API components and all external components that are wired from the algorithm, including file header, component parameters, nesC attributes, used and provided interfaces with their type arguments, instance parameters and instance names. Additionally, for all these interfaces the declaration of the commands and events is stored.

Since `wiring-check.xml` does not contain all the necessary information (e.g. how component parameters are employed in the interface list of configurations, or if a command or event is `async`) the files that contain the mentioned components and interfaces are additionally parsed by TinySwitch directly. Moreover, the source code of all internal components that need to be changed in the isolation layer because they wire external components is included in the XML file.

In summary, the XML file is self-contained and all necessary code can be generated in the next step using the resulting XML files only. Since the algorithms must not have dependencies on the application (see Wiring Requirements) a library of switchable algorithm sets can be built independently. When building the final application, a subset of a switchable algorithm set can easily be chosen to be included in the final binary.

## Example: Wiring Analysis of Sample Application

Our sample test application uses either the unified CTP or Flooding algorithm developed before. Compilation of both application versions results in the `wiring-check.xml`

files that are analysed further. For this purpose, the four API components of Figure 5.3 are declared as API components and the two other components of the unified interface as normal algorithm components. Additionally, the file paths “\$TOSDIR/lib/net/ctp/” and “\$TOSDIR/lib/net/4bitile” are configured to contain CTP’s internal components while two local files contain the Flooding implementation.

For CTP, the analysis process prints out several notes about the interfaces `Init`, `Pool`, `Queue`, `Timer` and `TaskBasic` in `LinkEstimatorP.nc`, `CtpForwardingEngineP.nc`, `CtpRoutingEngineP.nc` and `LruCtpMsgCacheP.nc`, as discussed before. Manual code inspection of `CtpForwardingEngineP.nc` reveals that `stop` could be improved by stopping the retransmission timer (which would post `sendTask` afterwards), emptying the send queue and putting the messages back in the message pool and by resetting other variables that reinitialise the component so that it starts cleanly when `start` is called later on:

```
command error_t StdControl.stop() {
    // added code for cleaner start/stop
    uint8_t i;
    fe_queue_entry_t *qe;
    // cancel current message that is being sent
    if (hasState(SENDING)) {
        qe = call SendQueue.head();
        call SubSend.cancel(qe->msg);
    }
    // stop retransmission timer
    call RetxmitTimer.stop();
    // empty send queue, notify sender and put back messages to pool
    while (! call SendQueue.empty()) {
        qe = call SendQueue.dequeue();
        if (qe->client < CLIENT_COUNT) {
            signal Send.sendDone[qe->client](qe->msg, FAIL);
        }
        else {
            call MessagePool.put(qe->msg);
            call QEntryPool.put(qe);
        }
    }
    // reset client queue entries - copied from Init.init()
    for (i = 0; i < CLIENT_COUNT; i++) {
        clientPtrs[i] = clientEntries + i;
    }
    seqno = 0;
    // reset internal state - replaces original
    // code that just resets ROUTING_ON state
    forwardingState = 0;
}
```

```

    // clear list of sent msgs
    call SentCache.flush();
    return SUCCESS;
}

```

In `LinkEstimatorP.nc` and `CtpRoutingEngineP.nc` some variables are set to their initial values in `stop`, too. Note again that these modifications are not absolutely necessary, but they provide for a much cleaner switch between the algorithms.

```

// LinkEstimatorP.nc
command error_t StdControl.stop() {
    // copied from variable definitions
    linkEstSeq = 0;
    prevSentIdx = 0;
    // copied from Init.init()
    initNeighborTable();
    return SUCCESS;
}

// CtpRoutingEngineP.nc
command error_t StdControl.stop() {
    running = FALSE;
    // code copied from Init.init()
    parentChanges = 0;
    state_is_root = 0;
    routeInfoInit(&routeInfo);
    routingTableInit();
    // copied from variable definitions
    sending = FALSE;
    justEvicted = FALSE;
    // original code continues
    dbg("TreeRoutingCtl", ...);
    return SUCCESS;
}

```

The `SplitControl` interface is used by both `CtpForwardingEngineP.nc` and `CtpRoutingEngineP.nc` to get notified about the status of the radio. Looking at the source code we note that CTP independently tracks the state of the protocol and the radio. Therefore, no specific order is required when powering on the algorithm and the radio. Since CTP builds on the radio we decided to first signal readiness of the radio and then to power on the algorithm, which corresponds to the *outside* action for the `SplitControl` interface.

Analysis for Flooding also suggests to check the `Pool`, `Timer` and `TaskBasic` interfaces of the Flooding implementation. However, no changes are necessary since the algorithm was already built with clean start/stop behaviour in mind.

### 5.2.3 Code Generation

When building an application with switchable algorithms, the engineer selects appropriate algorithms from a switchable algorithm set that have already been analysed, i.e. an XML file with analysis results exists for them. Using only these XML files, for each switchable algorithm set TinySwitch creates a multiplexing layer to direct application calls to the active algorithm and an isolation layer to block calls between a deactivated algorithm and the operating system.

It is advisable to put all generated files in a separate folder that is added to the **CFLAGS** in the make file of the application. For example, if the files are put in a folder “tinyswitch” directly underneath the application’s root folder, which also contains the make file, the line **CFLAGS += -I./tinyswitch/** is added to the make file. If a **CFLAGS** line is already present the additional **-I** switch has to be placed at first position. This ensures that the files generated by TinySwitch are found first and can override the original versions to include the modifications necessary for the isolation layer.

Finally, the new API components generated by the multiplexing layer have to be wired instead of a single algorithm, i.e. the **components** statements in the configuration have to be changed as explained in Section 5.2.1.

#### Multiplexing Layer

The code generator for the multiplexing layer gets only a list of XML files as input. Since the API components of different algorithms have different names, TinySwitch first tries to match the API components by their signatures, which include genericity, parameters and nesC attributes of the components, all interfaces and their direction (used/provided), type (command/event), type arguments, instance parameters and nesC attributes, and similar information for bare commands and events. If no match is found or if the match is not unique, the process stops. Such automated process saves the programmer from manual matching or from following special naming conventions. For example, in the algorithms of our sample application (see Figure 5.3) **RtCtpCtrlC** matches **RtFloodCtrlC** and **RtCtpSendRecvC** matches **RtFloodSendRecvC** from an application’s point of view.

For each set of matching components, a new configuration component for the multiplexing layer API is generated with the same signature as the respective components. This way, the multiplexing layer can be wired to the algorithm in the same way as one of the single algorithms. Inside, the configuration wires all interfaces and functions to a new corresponding internal module of the multiplexing layer, which in turn has several sub-interfaces in the opposite direction that are wired back to the original components in this set. For each command of a provided interface (and each provided bare command) and each event of a used interface (and each used bare event) of the API (i.e. the “incoming” functions of the algorithms), the internal module contains a **switch** statement that redirects calls and signals to corresponding sub-interfaces of



the currently activated original algorithm. Conversely, used commands or provided events that are called or signalled by the original algorithm components are only forwarded upwards via the multiplexer API if they come from the activated algorithm. For parameterised interfaces, TinySwitch also generates default commands and events since the application might only wire single instances, which would result in compiler errors without defaults.

Note that the multiple interfaces and the `switch` statement cannot be replaced by a parameterised interface in general to let nesC do the work and create the `switch` statements for us. Then, already parameterised interfaces would not be supported by the framework since nesC does not allow to wire one dimension of an interface. For example, the internal multiplexer module cannot provide `interface SubIf[uint8_t algo, uint8_t i]`, the algorithm use `interface If[uint8_t i]`, and they are wired with `MultiplexM.SubIf[1] -> Algo1.If` or similar.

TinySwitch also generates a small header file that contains the definition of a constant for each algorithm that is used in the switching layer to query if a specific algorithm is enabled or not.

### Example: Multiplexing Layer of Sample Application

TinySwitch creates the configuration `TC_RtSendRecvC` that is part of the new API components and that replaces `RtCtpSendRecvC` and `RtFloodSendRecvC`. Just like these, the new component is generic and can be parameterised with the AM id. The new configuration wires the provided interfaces to an internal module and the sub-interfaces of this internal module back to the original algorithm components:

```
generic configuration TC_RtSendRecvC(uint8_t _TC_Param1_) {
  provides interface Send;
  provides interface Receive;
}
implementation {
  components new RtCtpSendRecvC(_TC_Param1_);
  components new RtFloodSendRecvC(_TC_Param1_);
  components new TC_RtSendRecvM(_TC_Param1_);

  Send = TC_RtSendRecvM.Send;
  TC_RtSendRecvM.TCM_Sub_Ctp_Send -> RtCtpSendRecvC.Send;
  TC_RtSendRecvM.TCM_Sub_Flood_Send -> RtFloodSendRecvC.Send;
  // ... Receive omitted since similar to Send

  // connection to control interface - see below
  components TC_RtCtrlM;
  TC_RtSendRecvM.TC_Protocol -> TC_RtCtrlM.TC_Protocol;
}
```

## 5 TinySwitch - Switchable Components for TinyOS

The internal module contains the `switch` statements to reroute incoming calls to the enabled algorithm and `if` statements to block or forward outgoing calls:

```
generic module TC_RtSendRecvM(uint8_t _TC_Param1_) {
  provides interface Send;
  uses interface Send as TCM_Sub_Ctp_Send;
  uses interface Send as TCM_Sub_Flood_Send;

  // ... Receive omitted since similar to Send

  uses interface GetNow<uint8_t> as TC_Protocol;
}
implementation {
  // ... maxPayloadLength, getPayload, cancel of Send omitted

  command error_t Send.send(message_t*msg, uint8_t len) {
    switch (call TC_Protocol.getNow()) {
      case 0: // Ctp
        return call TCM_Sub_Ctp_Send.send(msg, len);
      case 1: // Flood
        return call TCM_Sub_Flood_Send.send(msg, len);
      default:
        return FAIL;
    }
  }

  event void TCM_Sub_Ctp_Send.sendDone(message_t*msg, error_t error) {
    if (call TC_Protocol.getNow() == 0) { // Ctp
      signal Send.sendDone(msg, error);
    }
  }

  // ... similar for TCM_Sub_Flood_Send.sendDone()

  // ... receive of Receive interfaces omitted
}
```

### Control Module

Each TinySwitch block needs a module to perform and control switching of algorithms. If an API component provides `StdControl` or `SplitControl` the TinySwitch control functions are added to the generated multiplexer module for this API component<sup>3</sup>.

---

<sup>3</sup>There must be only one API module providing `StdControl` and `SplitControl`.

Otherwise, a new module is generated. The reason for this is that during switching, **start** and **stop** commands of the algorithms to be switched are called if the algorithm API contains one of these control interfaces, and conversely the forwarding of these control interface commands depends on the internal state of the switching state machine.

The TinySwitch control module provides additional interfaces that are used to activate a new algorithm (interface **Set<uint8\_t>**) and to communicate the activation state of algorithms between this control module and the other internal modules of the multiplexing and isolation layer (interfaces **GetNow<bool>[uint8\_t]** and **GetNow<uint8\_t>**). Optionally, it can contain up to two control interfaces to fake **SplitControl** events for the isolation layer at different points of the switching process (see “Special Wiring Considerations” in Section 5.2.2). Additionally, the optional fake **StdControl** interface is implemented here which is used in the isolation layer instead of the real control interfaces of underlying components.

```
function Set.set(newAlgorithm)
  if currentAlgorithm != newAlgorithm
    if algorithm_is_started
      do
        optional: FakeInsideSplitControl.stopDone()
      do
        SubSplitControl.stop()
        while not successful
          wait for SubSplitControl.stopDone()
        while not successfully_stopped
          optional: FakeOutsideSplitControl.stopDone()
        end if
      currentAlgorithm = newAlgorithm
      if algorithm_was_stopped_above
        do
          optional: FakeOutsideSplitControl.startDone()
        do
          SubSplitControl.start()
          while not successful
            wait for SubSplitControl.startDone()
          while not successfully_started
            optional: FakeInsideSplitControl.startDone()
          end if
        end if
      end if
```

Figure 5.7: Pseudocode of command to initiate switching

Changing the active algorithm results in stopping the old one and starting the new one. Figure 5.7 shows the pseudocode for this process when using **SplitControl**

## 5 TinySwitch - Switchable Components for TinyOS

as the control interface. In short, the switching process constantly tries to stop the old algorithm until stopping was successful and then constantly tries to start the new algorithm until starting was successful. When `StdControl` is used, waiting for `startDone` and `stopDone` and the two outer loops enclosing these waits are omitted and `SubSplitControl` is changed to `SubStdControl`. Both `SubSplitControl` and the `FakeXXXSplitControl` interfaces apply to the currently active algorithm.

The `FakeXXXSplitControl` interfaces can fake `startDone` and `stopDone` signals of the underlying hardware if the algorithm needs such signals to work properly but the hardware was decoupled from the algorithms as explained in Section 5.2.1. In the isolation layer, these interfaces are wired to used `SplitControl` interfaces of internal algorithm components. This way, they get a “fake” signal that the hardware is on or off when the algorithm is started or stopped without actually starting or stopping the hardware.

Since loops and busy waits should be avoided in TinyOS the code switching functionality is actually implemented as state machine with a protocol state and a switching state. Instead of a loop a task is posted, which triggers reevaluation of the state. Figure 5.8 shows the complete state machine. The rounded rectangles represent the states with the name of the state in the top half and the optional actions performed on entering or leaving the state in the bottom half. The name of the state is composed of two parts: “*protocol\_x switch\_y*”, where *x* denotes the status of the algorithm and *y* the status of the switching process. Arrows leaving a state rectangle represent a transition, and the label describes the trigger for this transition and after a slash an optional action that is performed when the transition is taken. Small diamonds are choice points and their outgoing transition arrows are labelled with the necessary condition (called “guard”) for it.

In clockwise direction, the four states in the corners track the normal life cycle of an algorithm, which can be started and stopped using the `start` and `stop` commands of the `SplitControl` interface. From each of these states, calling `Set.set` initiates algorithm switching, which is represented by the states in the inner circle of the diagram. When no switching is in progress, the *protocol\_stopped...* and *protocol\_started...* states can be left when `start` or `stop` is called. In contrast, the switching process is self-executing: the `start` or `stop` commands are called automatically by two states and these states are left immediately. In all cases, the *protocol\_stopping...* and *protocol\_starting...* states wait for the `startDone` or `stopDone` notifications of the sub-control interface to go on. The different error codes resulting from these actions and events have to be observed carefully, which is more detailed in the state machine compared to the pseudocode (`EALREADY` is not explicitly handled there). Depending on whether the algorithm was started or stopped (or about to start or to stop) before entering the switching process, the new algorithm will be started or stopped after switching has been finished. For this, the additional variable `wasOn` is introduced to simplify the whole state machine.



## 5 TinySwitch - Switchable Components for TinyOS

We do not show code of our sample application here since the control module basically implements the state machine, the `switch` statements for the `StdControl` interface and some minor management code.

### Isolation Layer

The main task of the isolation layer is to forward calls from inner algorithm components to external components and callbacks from external components to inner components only for activated algorithms and block them otherwise. Additionally, it can change the behaviour of used `StdControl` and `SplitControl` interfaces. To achieve both, TinySwitch needs to change the wirings of the algorithm defined in internal algorithm configurations that have already been identified during wiring analysis. To do so, TinySwitch generates modified configurations with the same names that take precedence over the original files since they are placed in a directory that comes first in the list of include directories (see introduction to Section 5.2.3 for notes on the `CFLAGS` change).

The modified configurations are to a large extent a copy of the original files. However, all instantiations of components with interfaces to be isolated are replaced with instantiations of new configurations in the isolation layer. Also, control interfaces that should be “faked” are either rewired to a new instance of a `SplitControl` fake module (action *call*) or the `FakeXXXSplitControl` (actions *inside* and *outside*) or `FakeStdControl` (action *cut*) interfaces of the control module.

The new configurations, which are also generated by TinySwitch, have the same genericity, component parameters, nesC attributes and interfaces as the components that originally provided the interfaces to be isolated. This is necessary to allow for the described exchange of components. In the implementation of these configurations, the original component is instantiated and all interfaces that do not need isolation (see Section 5.1.4) are just “wired through”. All interfaces to be isolated are routed to new modules of the isolation layer and from there to the original component.

These new interface-specific modules forward commands or events only if the algorithm is active. Otherwise, a specified default value is returned. The activity information is queried from the control module using the `GetNow<bool>` interface. The module needs to be generic since it is instantiated once per isolated interface per component (note that a component can use several interfaces of the same type with just different names).

### Example: Isolation Layer of Sample Application

All connections that need rewiring for CTP are implemented in `CtpP.nc`. There, 7 lines are modified by the framework. The original components `AMSenderC`, `AMReceiverC` and `AMSnooperC` are replaced by `TCS_Ctp_AMSenderC`, `TCS_Ctp_AMReceiverC` and

TCS\_Ctp\_AMSnooperC. As explained before, these automatically generated configurations have the same genericity and parameters as the the original components.

The following code snippet shows two original lines of the CtpP.nc configuration. The first line declares a new instance of **AMSenderC** (external component) and the second line wires its **AMSend** interface to **Forwarder** (internal component). The line that replaces the component declaration is also shown: a component of the isolation layer is instantiated and named as **AMSenderC**. Thus, no further changes are necessary. The same happens for **AMReceiverC** and **AMSnooperC** (not shown).

```
// original content of CtpP.nc
components new AMSenderC(AM_CTP_DATA);
Forwarder.SubSend -> AMSenderC;

// replacement line for CtpP.nc
components new TC_Ctp_AMSenderC(AM_CTP_DATA) as AMSenderC;
```

Since it was decided during wiring analysis (see example in Section 5.2.2) to fake radio readiness events this rewiring also happens in CtpP.nc:

```
// original content of CtpP.nc
Router.RadioControl -> ActiveMessageC;
...
Forwarder.RadioControl -> ActiveMessageC;

// replacement line for CtpP.nc
components TC_RtCtrlM;
Router.RadioControl -> TC_RtCtrlM.FakeOutsideSplitControl[_TC_RT_CTP];
...
Forwarder.RadioControl
  -> TC_RtCtrlM.FakeOutsideSplitControl[_TC_RT_CTP];
```

Only the interfaces **AMSend** and **Receive** need to be isolated, all other interfaces are directly wired to the original components. This is reflected in the following replacement configuration for **AMSenderC**. Only **AMSend** is rewired to an internal module of the isolation layer. This module also gets a connection to the control module to query the activation state of the algorithm:

```
generic configuration TCS_Ctp_AMSenderC(am_id_t _TC_P1_) {
  provides interface AMSend;
  provides interface AMPacket;
  provides interface Packet;
  provides interface PacketAcknowledgements as Acks;
}
implementation {
  components TC_RtCtrlM;
  components new AMSenderC(_TC_P1_);
```

## 5 TinySwitch - Switchable Components for TinyOS

```
AMPacket = AMSenderC.AMPacket;
Packet = AMSenderC.Packet;
Acks = AMSenderC.Acks;

components new TCS_AMSendM();
AMSend = TCS_AMSendM.AMSend;
TCS_AMSendM.TC_Sub_AMSend -> AMSenderC.AMSend;
TCS_AMSendM.TC_IsEnabled -> TC_RtCtrlM.TC_IsEnabled[_TC_RT_CTP];
}
```

The new TCS\_AMSendM module of the isolation layer finally contains the code to forward the commands or events if the algorithm is enabled:

```
generic module TCS_AMSendM() {
  provides interface AMSend;
  uses interface AMSend as TC_Sub_AMSend;
  uses interface GetNow<bool> as TC_IsEnabled;
}

implementation {
  command error_t AMSend.send(am_addr_t addr, message_t*msg,
                             uint8_t len) {
    if (call TC_IsEnabled.getNow()) {
      return call TC_Sub_AMSend.send(addr,msg,len);
    }
    return FAIL;
  }

  event void TC_Sub_AMSend.sendDone(message_t*msg, error_t error) {
    if (call TC_IsEnabled.getNow()) {
      signal AMSend.sendDone(msg,error);
    }
  }

  // maxPayloadLength, getPayload, cancel similar to send()
}
```

The TCS\_AMReceiveM module is similar. Both TCS\_AMSendM and TCS\_AMReceiveM are only specific to the interface and can also be used in the isolation layer of the Flood algorithm. Therefore, they need to be generic so that for each isolated interface a new isolation instance can be instantiated.

The Flooding algorithm is changed in a very similar way. Therefore, no code is shown here.



### Example: Complete Sample Application

Eventually, TinySwitch has created the multiplexing and two isolation layers, each of which for CTP and Flooding, respectively. Table 5.1 summarises the generated code by TinySwitch for this sample application. “SLOC” denotes the source lines of code, i.e. the source code lines without comments and blank lines. The “new SLOC” line for the Multiplexing Layer also includes the management code, e.g. the switching state machine.

Multiplexing Layer		
new SLOC		258
new files		5
Isolation Layer		
	CTP	Flooding
new SLOC	98	79
new files	5	4
changed SLOC	7	1
changed files	1	1

Table 5.1: Overview of generated code for switching of routing algorithms

The final architecture of the generated switching solution is shown in Figure 5.9. `TC_RtCtrlC` and `TC_RtSendRecvC` are the new API to be used in the application. `TC_RtCtrlM` and `TC_RtSendRecvM` perform the multiplexing operation. Additionally, the control module `TC_RtCtrlM` provides activation state information to other internal modules of the multiplexing and isolation layer. The isolation layer disconnects the `AMSend` and `AMReceive` interface from the operating system.

## 5.3 Conclusion

We have shown TinySwitch, a generic framework for run-time algorithm switching. After the programmer has ensured that all alternative algorithms (e.g. all routing algorithms) share a common API, TinySwitch analyses the algorithms and suggests optimisations for better switching performance. When building the final application, the engineer can select appropriate algorithms and all necessary switching code for TinyOS is automatically generated by TinySwitch. Since the analysis of the algorithms is decoupled from the actual code generation process a library of switchable algorithms can be built independently.

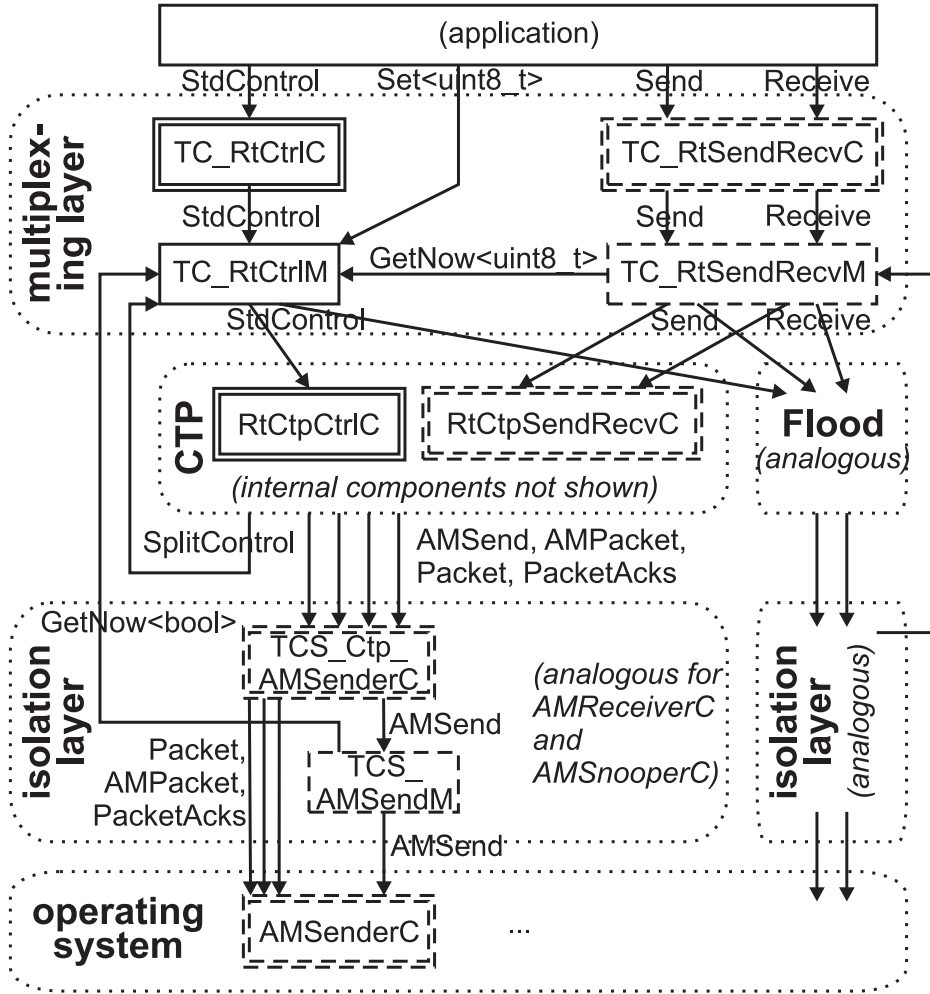


Figure 5.9: Multiplexing and isolation layers for routing algorithms CTP and Flooding

In contrast to many other code exchange approaches TinySwitch avoids flash reprogramming but switches algorithms already included in the application binary. It ensures that only a single algorithm for the same task is active at a time and isolates all others. Moreover, TinySwitch automatically generates all necessary code and requires only minimal work if algorithms are already implemented cleanly. Of course, TinySwitch is generic and not tailored to a certain algorithm class, and it has only minimal requirements to the algorithms to be used with it. To ensure maintainability, TinySwitch runs as standalone software during the development process and does not require any modifications to the TinyOS make chain or the nesC compiler.

While TinySwitch can be used as a standalone algorithm switching solution, the aforementioned features together with its fast switching time make TinySwitch perfectly meet the requirements of algorithm exchange for adaptation purposes and, thus, for TinyAdapt.

# 6 Evaluation

This chapter evaluates the generic adaptation framework TinyAdapt as proposed in Chapter 3 using the TinyOS implementation shown in Chapter 4 as well as the switching framework for TinyOS, TinySwitch, presented in Chapter 5. Also, the mobility metric and the monitoring system are evaluated.

In general, an evaluation of architectures and procedures is difficult since their results can also be obtained by manually writing code that implements mostly similar ideas as in the general approaches. However, the effort to write this code compared to the effort using frameworks depends on the ability and knowledge of the user. For these reasons, we mostly present case studies that use TinyAdapt for parameter based adaptation (Section 6.4), use plain TinySwitch to include two MAC algorithms in an application and switch between them from the application layer (Section 6.5) and finally use TinyAdapt with TinySwitch for autonomous adaptation that switches between two routing algorithms (Section 6.6). In all case studies we show that the use of TinyAdapt and/or TinySwitch is beneficial compared to non-adaptive or non-switchable solutions and that, therefore, our frameworks are a effective way to create such applications.

Finally, we analyse the overhead of TinyAdapt and TinySwitch in terms of memory usage and processing time. Also, the effort for node reprogramming is shown which is avoided when TinySwitch is used.

## 6.1 Experimental Setup

All experiments in this chapter are performed using Avrora for simulations and the TWIST testbed for experiments on real nodes.

Avrora [TLP05] emulates a ATmega128 microcontroller and allows to run real AVR binaries with precise timing. All other devices available on MicaZ nodes such as radio chip, serial flash and sensors are simulated according to their specification. Thus, TinyOS programs compiled for the MicaZ platform can be run with Avrora. The emulation and simulation includes also energy calculations based on the current operational state of the different devices.

When simulating a complete sensor network, radio links between the nodes are simulated with either a unit-disc graph or a lossy radio model. We use the lossy model,

which considers fading, noise and interference. Avrora also supports mobile nodes: the Random Waypoint mobility model is used for mobile scenarios unless otherwise stated.

Avrora comes as a single Java application, which is suitable for massive parallel evaluation runs in compute clusters. Since the experiments were conducted over a longer time period, different Avrora versions (1.7.109–1.7.115) were used. Since 2009, Avrora has also been maintained actively by the Networked Embedded Systems group of the University of Duisburg-Essen.

In the TWIST testbed [HKWW06] over 100 TmoteSky (TelosB) nodes are deployed over three floors in offices at the TU Berlin campus. No exact node locations are provided due to privacy reasons.

TWIST is accessible via a web interface where an experimentation slot needs to be reserved for the user. During this slot, the user has exclusive access to the testbed. Via the web interface, binary programs can be uploaded, nodes can be switched on and off, and the logging of data written to the nodes' serial ports can be turned on and off. After the experiment, this logging data can be downloaded and evaluated off-line. All these control actions can also be scripted using *wget* or *curl*, which allows for automatic evaluation of WSN software.

Since the nodes are installed statically no mobile scenarios can be evaluated with TWIST. Also, TWIST does not provide for energy measurements of the nodes. For both, simulations on Avrora have to be used.

## 6.2 Mobility Metric

In this section, we exemplarily show how to configure the mobility metric and the monitoring component and we present the resulting metric values. Since no mobile testbed with a larger number of mobile nodes was available this evaluation is based on simulation using Avrora 1.7.115.

We start with determining the beacon rate of the mobility metric using the formulas of Section 4.4. With TinyOS' default transmission power setting, packets are received with high probability when the node distance is not larger than  $r = 14m$ . For a node speed of  $v = 4m/s$  the relative node speed is about  $\tilde{v} \approx 4v/\pi \approx 5.1m/s$ . If we set the packet interval to  $T_p = 5s$  the travelled distance during this interval of  $d = T_p \tilde{v} = 25.5m$  is smaller than twice the transmission radius, and approximately 84% of all packets can be detected by neighbour nodes.

As advised in Section 4.2 we conducted an experiment where each node sends periodic beacons and locally measures the link changes and the number of neighbours. Between 20 and 100 nodes move with fixed speeds ranging from 1 to  $6m/s$  in a  $100 \times 100m$  area. For each combination, we ran 10 simulations. The mean value and empiric standard deviation for the number of neighbours and the link changes were calculated and used

with the formulas described in Section 4.4. Figure 6.1 shows the predicted 95% ( $\mu \pm 2\sigma$ ) ranges of the “link changes per number of neighbours” metric values for different values of the exponential moving average (EMA) smoothing factor  $\alpha$ .

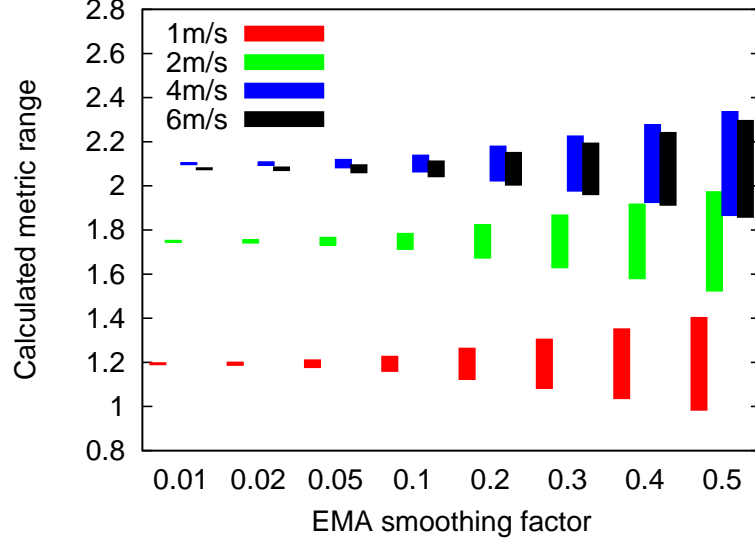


Figure 6.1: Calculated ranges of the metric values for different  $\alpha$

The values for the movement speeds of  $4m/s$  and  $6m/s$  are very close even for very low  $\alpha$  values due to the following two reasons. First, the packet interval is too long since it was set for a maximum speed of  $4m/s$  and the metric is thus severely affected. Second, effects of the bounded area become prominent since a fast node moves out of range but returns soon after reaching its intermediate destination and, therefore, no time-out occurs. However, for velocities up to and including  $4m/s$  the metric values are clearly distinguishable with 95% probability up to an EMA smoothing factor  $\alpha = 0.3$ . Therefore, this value is used in the following.

To evaluate the applicability of our metric, we used the measurements of our previous simulations and calculated the final metric with  $\alpha = 0.3$  every 20 seconds as the nodes would also do in reality. Figure 6.2 shows the average as well as the 3rd and 97th percentiles as error bars. It can be noticed that the absolute metric value is practically not affected by the total number of nodes. However, the variance is higher for a lower number of nodes so that the metric values exceed the predicted range.

In TinyAdapt, adaptation is not based on the average of the mobility metric values but on its network-wide maximum. Therefore, we also calculated this maximum at each point in time. Note that this is the ideal maximum, which is only available from a global perspective. In a sensor network, the maximum will be calculated by the monitoring component, which will be evaluated in the next section. Again, Figure 6.3 shows the average as well as the 3rd and 97th percentiles as error bars. Still, the velocities can be distinguished well with only minor overlapping between 2 and 4 m/s for 20 nodes. Nevertheless, deeper analysis shows that the peaks occur only for a short time and will be overridden quickly.

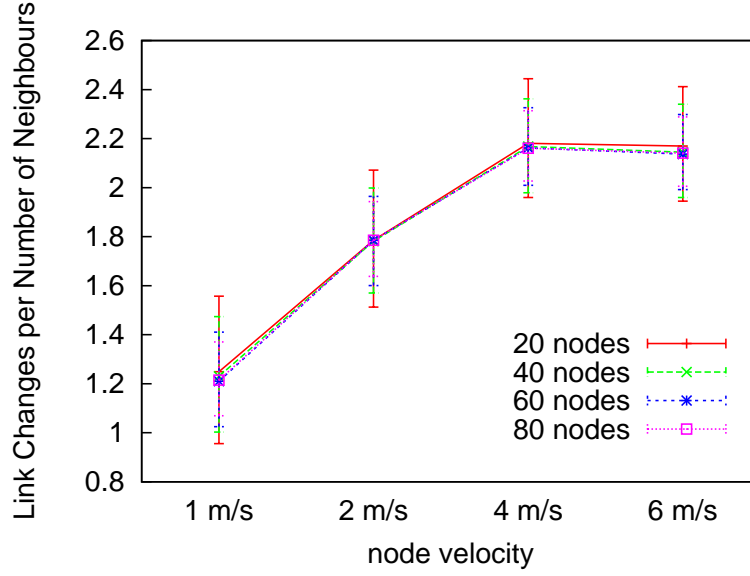
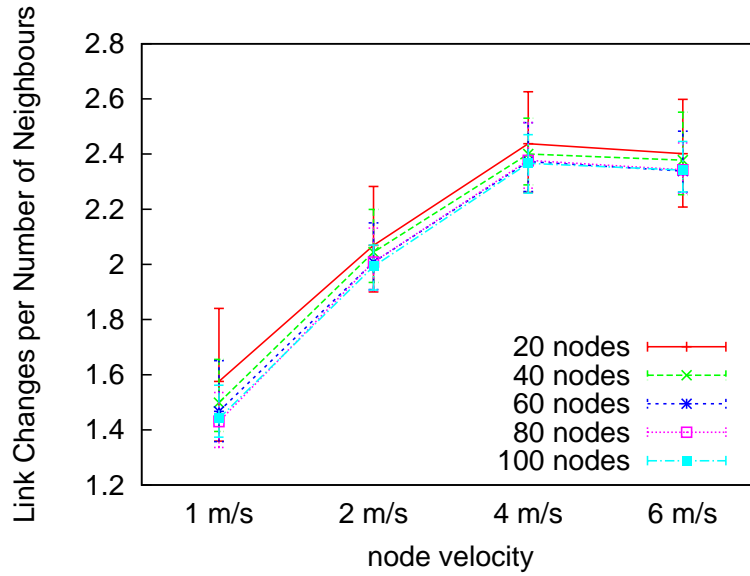
Figure 6.2: Real value range of the final metric with  $\alpha = 0.3$ 

Figure 6.3: Value range of the metric's global maximum

Overall, the “link changes per number of neighbours” metric is a very good indicator for the node speeds if the smoothing factor is configured properly. Therefore, it can also be used to drive adaptation as will be shown later in Section 6.6.1.

## 6.3 Monitoring System

The monitoring system presented in Section 4.3 is used to calculate a network-wide maximum of the network metrics upon which the adaptation is based. Therefore, the most crucial issue is the convergence time, i.e. the time required until all nodes in the network agree on a new maximum of the metric. This property is explored in the following based on Avrora simulations.

To confirm the assumption of Section 4.3 that in static networks the convergence time depends on the network diameter, we arranged 25 nodes in a square grid with a distance of  $14m$ , which allows for communication with the horizontally and vertically adjacent neighbours. This results in a network diameter of 8 hops from one corner to the opposite corner. All nodes choose a random local value every  $60s$ . We measured the duration between the point in time at which the last node sets a new local value and the point in time at which all nodes have converged to the new maximum.

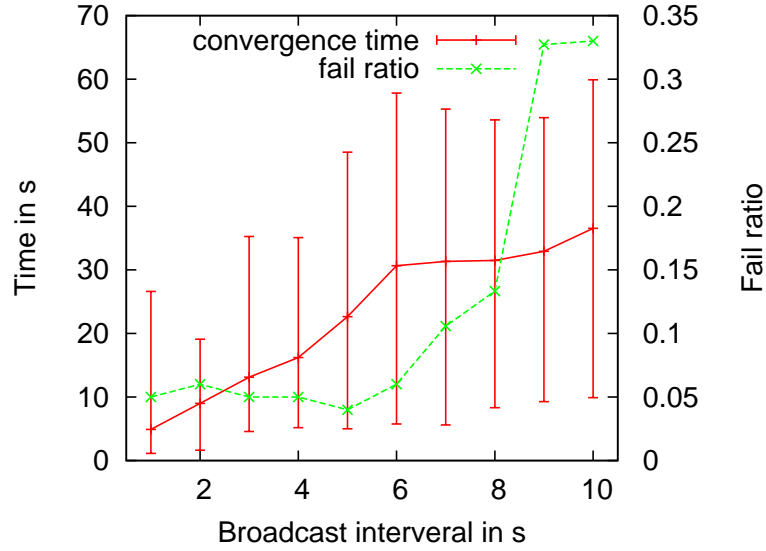


Figure 6.4: Convergence time and the fail ratio with various broadcast intervals

The solid line in Figure 6.4 shows the convergence time (average with minimum and maximum as error bars) for different intervals at which each node broadcasts its current maximum. It includes only the measurements when the network succeeded in converging. The dashed line depicts the ratio of iterations in which the values failed to converge. Clearly, the convergence time increases with increasing broadcast intervals. However, the fail ratio raises suddenly when the value of the broadcast interval times the hop count to reach all nodes becomes too large, which is at  $60s/8 = 7.5s$  in our scenario.

To show the dependency of the velocity and mobility models, we placed 25 nodes randomly on a  $100 \times 100 m^2$  area to ensure minimum connectivity and made them moving using both the Random Waypoint and the Manhattan mobility model with different

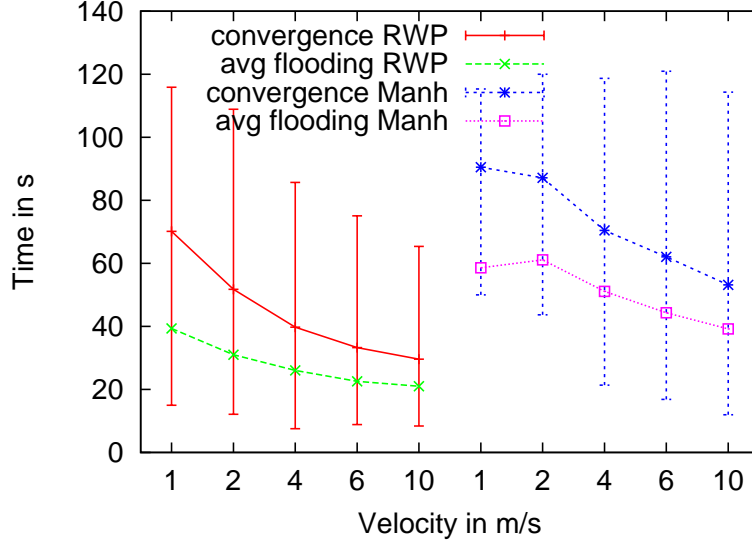


Figure 6.5: Convergence time and average flooding time for different velocities with Random Waypoint (left) and Manhattan (right) Model

velocities. The broadcast interval was set to 5s (to be equal to the determined beacon interval of the mobility metric) and all nodes chose a random local value every 120s. Figure 6.5 shows the average flooding time, i.e. the time after which a node knows the correct network-wide maximum, and the minimal, average and maximum convergence time, i.e. the time after which the whole network agreed to this maximum.

For the Random Waypoint model, both values converge to the times measured in the static setting since mobility leads to fast exchange with distant locations. The values for the Manhattan model are higher since nodes do not often cross the centre of the area, but convergence can still profit from higher velocities. In general, the convergence time lies approximately 50% over the flooding time, which is needed to reach the last node from the middle of the network.

Figure 6.6 analyses the situation when the network fails to converge within the 120s limit. With the Random Waypoint model, even with very low mobility, which leads to longer-lasting partitioning, the network did not converge to the maximum in only 28% of the cases. With higher mobility, the fail ratio drops to approximately 10%. This does not affect all the nodes: on average 4–9 nodes have not received the maximum, and this number drops with higher velocity. For the Manhattan model, the fail ratio is very high with low mobility since the partitions can last much longer.

However, the fail ratio and the number of nodes not receiving the maximum are less serious when we use the monitoring component to calculate the maximum of the locally measured metric value since this value does not change arbitrarily. Therefore, we now combine the mobility metric with the monitoring component to calculate a maximum of ever-changing base values. Of course, this leads to an ever-changing maximum value as well. Figure 6.7 shows the results of 10 simulations with a fixed velocity of 1m/s over



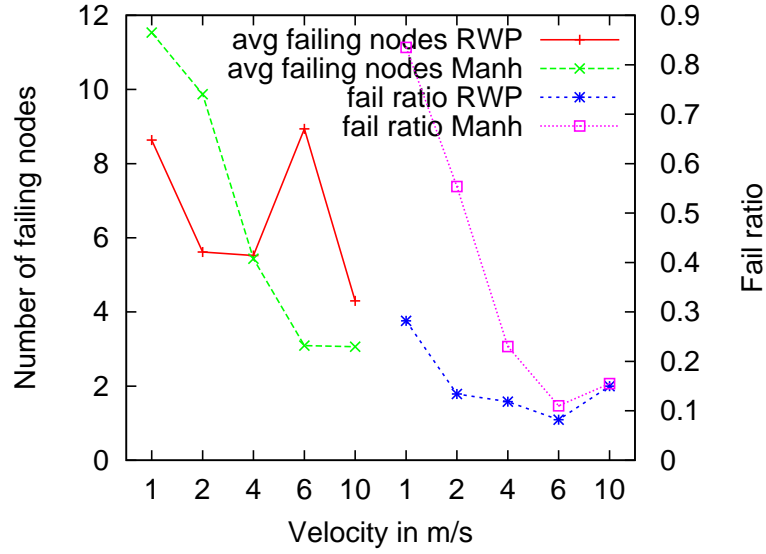


Figure 6.6: Average number of failing nodes (left) and fail ratio (right) for different velocities with Random Waypoint and Manhattan Mobility Models

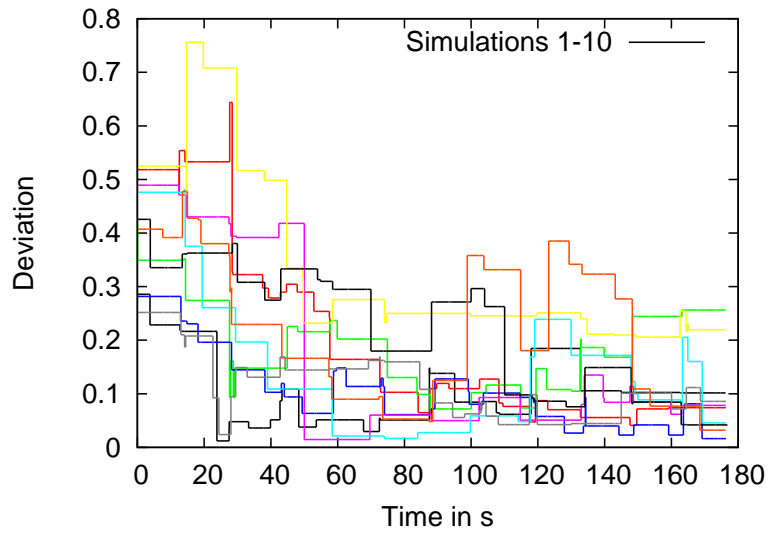


Figure 6.7: Maximum deviation from network-wide maximum over time

time. Each line plots the maximum error computed as the absolute deviation from the current network-wide maximum in each simulation. That is, for any given deviation, there is at least one node that deviates by the indicated value from the network-wide maximum. The error is large at the beginning but drops below 0.2 quickly due to the propagation of the network-wide maximum. However, the whole network does not converge completely due to the fast changing metric values that provide the local value and, thus, there is always a remaining error. Although potentially problematic, this is not an issue as long as the maximum error is not constantly occurring at a single node, which is not the case both due to the exponential moving average and due to the

exchange with other nodes. Therefore, the metric stabilisation time was introduced before performing adaptation as described in Section 4.7.

In summary, the monitoring framework achieves calculation of the network-wide maximum of a network metric. Due to the properties of the metrics and the monitoring process the network-wide maximum is quite stable, exhibits low deviation and is, therefore, suitable to drive adaptation.

### 6.4 Case Study: Parameterisation based on User Preferences

In TinyAdapt, adaptation can be triggered by changing user preferences or by changing network conditions, and adaptation can be performed by parameterisation or exchange of the algorithm. In our first case study we show how TinyAdapt performs adaptation by algorithm parameterisation when the user preferences are changed through new Goal Definitions.

We consider a security monitoring scenario where an area is to be monitored for a longer time without further intervention. In case a suspicious event is detected the security office can decide to select a higher data rate with a more robust transmission scheme. Of course, this will consume more energy than the long-term observation scheme and is, therefore, not suitable for normal operation. We will show that, using TinyAdapt, this change of high-level user preferences will lead to a low-level adaptation that satisfies both schemes in an optimal way.

Since in this case study a static scenario and parameter-based adaptation is considered only, no Monitoring and Installation components are used, neither in the pre-installation nor in the run-time phase. They will be covered in later case studies.

#### 6.4.1 Algorithm Modification and Application

To perform the security monitoring, each node regularly sends measurements to a base station. Since we are not interested in the actual measurements for this evaluation we abstract from the sensors and only send dummy values in the form of a counter.

We have built our application on TinyOS 2.1.0 using the Collection Tree Protocol (CTP) [FGJ<sup>+</sup>07]. We changed the original TinyOS implementation of CTP so that the maximum beacon interval and the maximum number of retries for sending a data packet from node to node are stored in TinyXXL. Additionally, we enabled Low Power Listening (LPL) of TinyOS' BoX-MAC [ML08] and store the LPL interval in TinyXXL as well. We also added `dataChanged` events to get notified when one of these values is changed to take further action. For example, the main module needs to stop and

restart its packet interval timer or the LPL integration module must set the new local wakeup interval of LPL. This was partly shown as example in Section 4.1.2.

### 6.4.2 Algorithm Exploration

This case study is based on simulations using Avrora 1.7.109 with some additional bugfixes. In our test scenario, 25 MicaZ nodes are placed in a 5x5 grid. The base station is located in one corner of this grid. In general, communication is possible between the horizontal and vertical neighbours of a node. No further parameters are necessary for this static scenario. Moreover, the characteristics of this small setup are well known so that no configuration application needs to be run.

In each simulation, nodes are booted randomly at the beginning. After 5s setup time each node starts sending data packets regularly for 120s. Then, after another 5s to allow for in-transit packets to be delivered to the base station, the simulation ends.

Four different algorithm parameters are tested as shown in Table 6.1. The values for the data interval are chosen according to possible application needs. The standard value for the maximum number of retransmissions in CTP is 30 but since that many retransmissions occur rarely and a reduction might save energy we select smaller values inspired by binary search. The standard value for the maximum CTP beacon interval is 512s but we select smaller values in the same way as before since they might make the network more robust. Finally, for LPL we test without LPL and with different wakeup intervals of several milliseconds since we assume that higher values might save more energy. It has to be noted again that this assessment is black box testing of an algorithm; therefore some assumptions will not hold as can be seen later.

Algorithm Parameter	Tested Values
data interval of application	1, 2, 10, 30 s
max. number of CTP retransmissions	2, 8, 15, 30
max. CTP beacon interval	32.77, 128, 512 s
low power listening wakeup interval	LPL off, 250, 500, 1000 ms

Table 6.1: Algorithm parameters for security monitoring application

As performance metrics, we select energy consumption as general important metric of sensor networks, and delivery ratio and latency as two metrics characterising routing efficiency. A brute-force exploration of all parameter combinations is done (see Section 4.5), and the following results are based on 10 simulations each. They are presented in detail and analysed manually to be able to understand the adaptation results in the next section, but we do not try to completely explain the behaviour of the algorithms since they are treated as black boxes by the adaptation engine.

Analysis shows that the maximum beacon interval has very little influence on the performance parameters: when keeping the other input parameters constant and changing

## 6 Evaluation

the maximum beacon interval the average difference between the best and the worst measure is 2.1 percentage points for the delivery ratio,  $1.26J$  for the total network energy and  $1.17s$  for the average maximum latency. Therefore, this parameter can be regarded as irrelevant and is set to its default value of  $512s$  for the rest of experiments.

A similar observation can be made for the maximum number of retransmissions: for values of 8, 15 and 30 the average difference between the best and the worst measure is 1.7 percentage points for the delivery ratio,  $1.01J$  for the total network energy and  $1.35s$  for the average maximum latency. Therefore, we remove the values of 8 and 15 from the parameter space of the maximum number of retransmissions and keep 30 since it shows a slight advantage over the other values. On the other hand, limiting the number of maximum retransmissions to 2 leads to low delivery ratios for longer packet intervals. Since less energy is needed for less retransmissions this setting might be relevant when the delivery ratio does not matter. Therefore, we keep it as well. Note that automatic Configuration Table post processing (Section 4.6) would show similar results.

The influence of the remaining two parameters data packet interval and LPL interval is shown in Figure 6.8 for the delivery ratio, Figure 6.9 for the energy and Figure 6.10 for the latency.

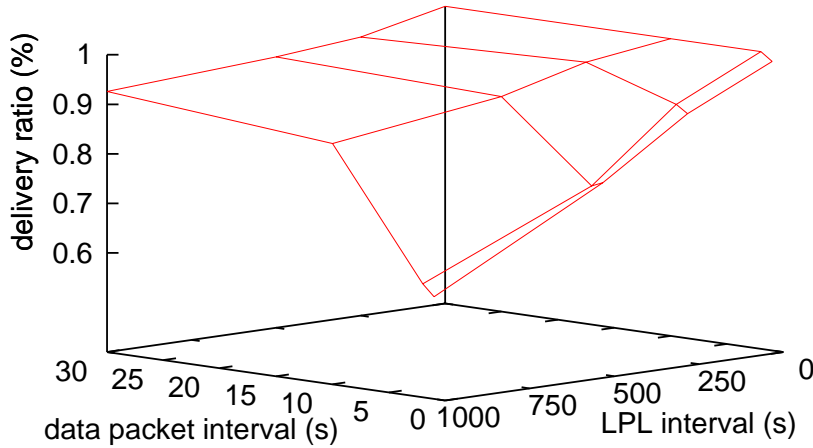


Figure 6.8: Influence of data packet and LPL interval on the delivery ratio for Security Monitoring Application

With increasing LPL interval, the delivery ratio decreases since the time in which LPL tries to repeatedly send a packet is longer, thus disturbing other transmissions for a longer timer. Interestingly enough, energy does not decrease either, but the LPL interval of  $250ms$  consumes the least energy. Obviously, the maximum latency increases with increasing LPL interval since a node has to wait the interval time in the worst case for each transmission attempt until the neighbour is listening again.

Not surprisingly, a higher load (i.e. a lower packet interval) increases the energy and decreases the delivery ratio since collisions occur more often. When using a

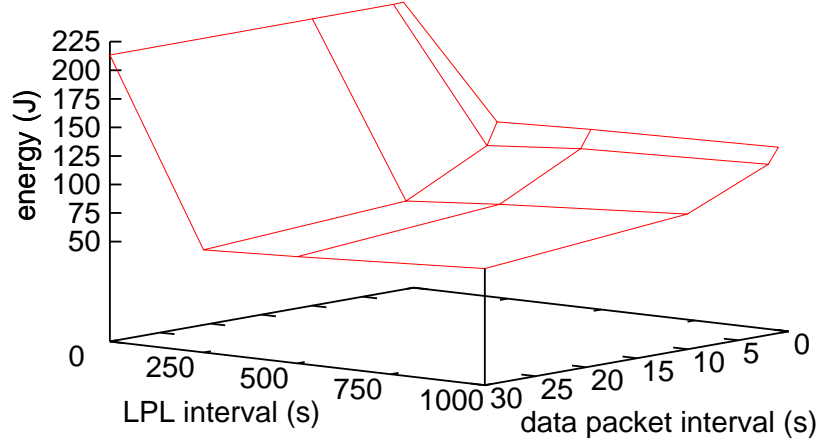


Figure 6.9: Influence of data packet and LPL interval on the energy for Security Monitoring Application

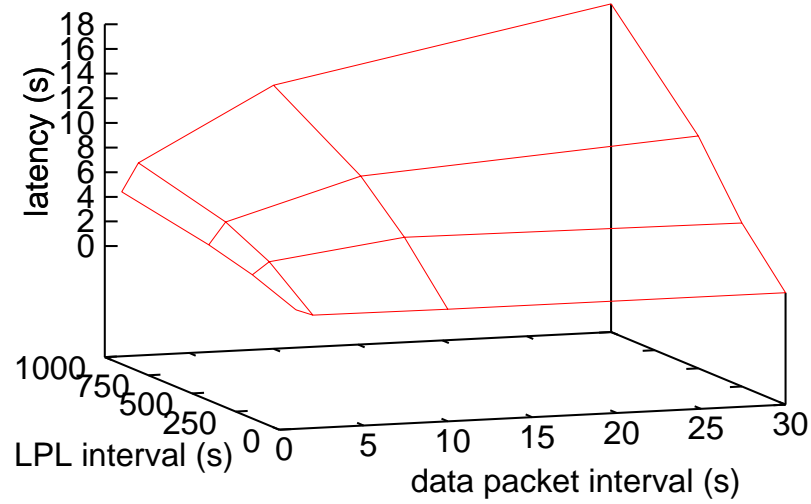


Figure 6.10: Influence of data packet and LPL interval on the latency for Security Monitoring Application

data interval of 1s or 2s together with higher LPL intervals, the delivery ratio drops significantly, thus indicating an overload. When LPL is switched off (LPL interval = 0) CTP is more robust to higher load. The maximum latency decreases with lower packet intervals since the sending nodes are often waiting for a free channel and start sending before the receiving node switches to sleep mode.

We have now analysed the routing algorithm parameterisation in detail. In the next section we will show how these results are used by our adaptation engine to optimise the behaviour of the application.

### 6.4.3 Run-time Results

When building the adaptive application the Goal Definition for the starting operation mode is evaluated and the resulting parameters are set as default. In our example of a long-term observation application the Goal Definition requires a minimal delivery ratio of 90% and advises to optimise energy, which leads to a data packet interval of 30s and a LPL interval of 250ms.

Simulation starts as before. After 120s, the user decides to switch from the long-term observation mode to a fine-granular observation. Therefore, a new packet interval of 2s and a maximum latency of 1s as new requirements and delivery ratio as new optimisation goal are distributed to the network via the base station. After a node has received a new Goal Definition, TinyAdapt waits for 5s to enable further dissemination before performing adaptation and configuring the algorithms accordingly. In the described case, it will set the data interval to 2s as requested by the Goal Definition and it will switch off low power listening since only then the latency constraint can be met. The maximum number of retransmissions is kept at 30 since this will result in the best delivery ratio. Finally, the simulation stops after 260s.

In the presented application, the Configuration Table needs 448 bytes of RAM, which can be further reduced since we did not remove parameters that remain the same for all the table entries. The new Goal Definition that is distributed in the network is only 13 bytes long. The adaptation process, i.e. the selection of the best entry in the given Configuration Table according to the given Goal Definition, takes 1.32ms only.

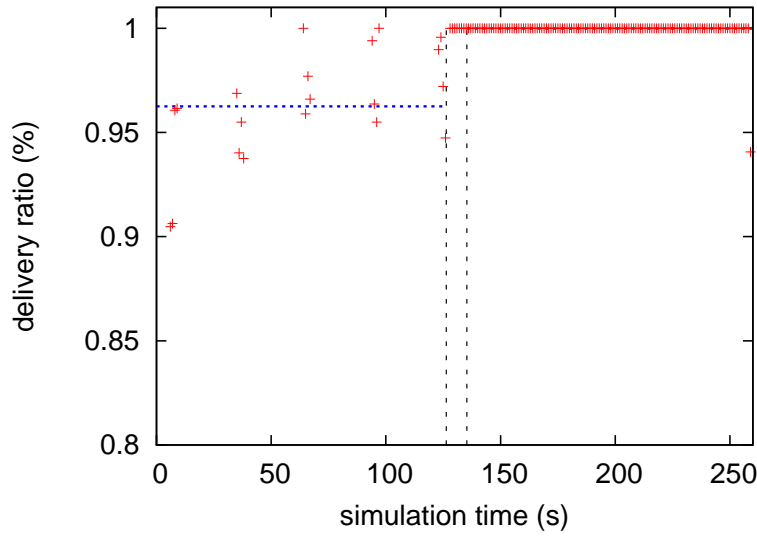


Figure 6.11: Delivery ratio over time for adaptive security monitoring application

The results of 25 simulation runs are shown in Figures 6.11, 6.12 and 6.13. The two vertical lines always indicate the time span between the adaptation of the first node and the last node of all simulations. Figure 6.11 shows the delivery ratio of all packets sent in a 1s interval. The horizontal line indicates the value obtained from

the basic simulations. The actual 1-second ratios are distributed around this line, but the mean value of 95.9% is only slightly worse than the prediction of 96.25% from the basic simulations. After the nodes have adapted, all packets are received by the base station. Only in the last timeslot the delivery ratio drops to 94.06% but it must be noted that in the simulation of the adaptive scenario the nodes do not stop creating new packets after a certain time like for the basic simulations and, therefore, packets might still be in transit when simulation ended.

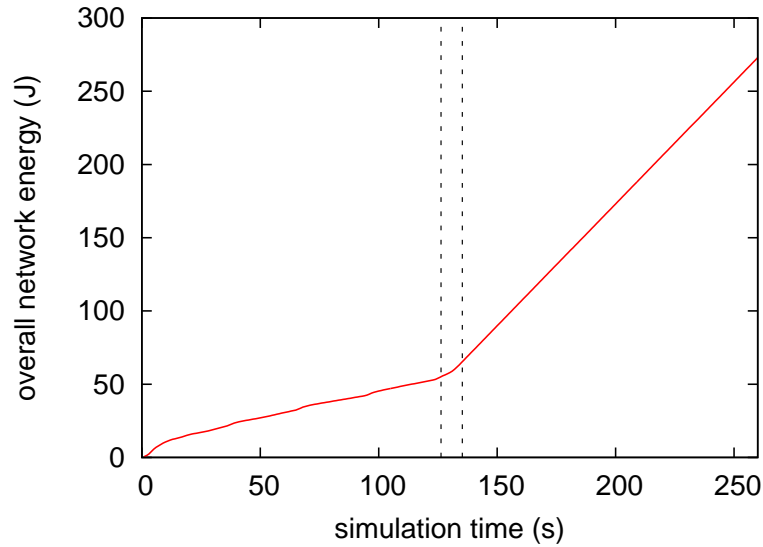


Figure 6.12: Overall network energy over time for adaptive security monitoring application

Figure 6.12 shows how the overall network energy is developing over time. The basic simulations predict a needed energy of  $52.22J$  for the first 130s. Since the adaptation starts approximately 4s earlier and since the adaptive application also includes Drip (for disseminating the Goal Definition, see Section 4.1.1) the needed energy is a little higher ( $57.88J$ ). For the next 130s,  $215.07J$  are required which is only slightly more than estimated  $213.71J$  according to the basic simulations. When LPL is off, there is almost no difference if the radio is receiving or sending. Therefore, the additional Drip component has only a minor influence on the overall energy.

In Figure 6.13 the maximum latency of all packets sent in a 1s interval is shown. The two horizontal lines represent the latency values from the basic simulations for the currently active algorithm parameter combination. As can be seen from the graph these values well establish an upper bound for latency for all transmissions; only twice the bound is exceeded minimally.

In total, TinyAdapt succeeds in parameterising a running application according to user preferences, which can be changed during run-time, in such a way that the application meets the defined goals.

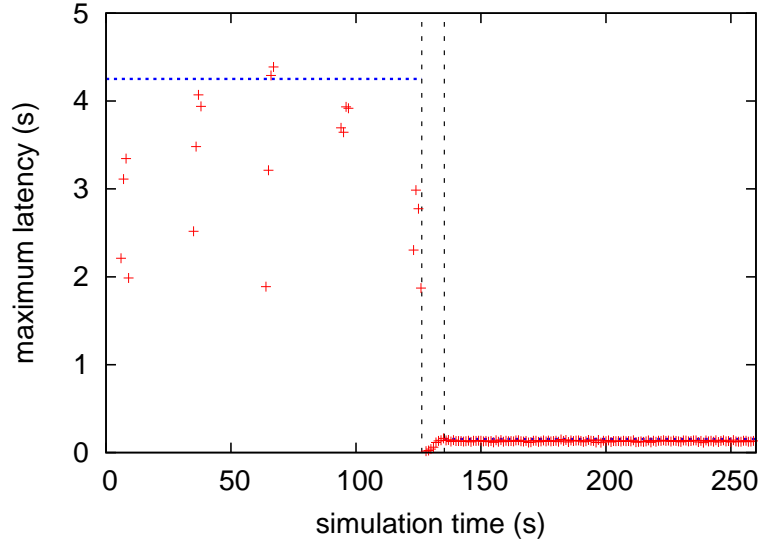


Figure 6.13: Latency over time for adaptive security monitoring application

## 6.5 Case Study: Switching of MAC Algorithms

Before evaluating adaptation using algorithm exchange/switching, we evaluate TinySwitch as standalone solution. Therefore, in our second case study, the application layer directly uses TinySwitch to switch between suitable MAC algorithms. We compare this TinySwitch application to a non-adaptive application and to one performing manual parameter-based adaptation.

As application scenario, we consider wild horse monitoring in the Doñana Biological Reserve, one of the main tasks in the PLANET project [PLA]. There, the horses are moving freely in a large area most of the time, but they regularly come to a few waterholes. Motes on the collars attached to the horses regularly store their GPS location, accelerometer and magnetometer data to serial flash memory and transmit the collected data to the base stations installed at the waterholes. Additionally, the motes regularly send radio beacons containing their locations, which are received and also stored by neighbouring nodes to approximate the locations of horses that do not come closer to the waterholes themselves. Using this data, social and mobility patterns shall be inferred and important research on super-spreaders of diseases [Ste11] shall be conducted.

To save energy, low power listening (LPL) with the standard TinyOS BoX-MAC protocol is used when transmitting the normal beacons. However, due to the number of horses being in the radio range of the base station normal CSMA-based channel access performs poorly due to high collisions, even without LPL. Here, a TDMA-based MAC protocol with the base station as a coordinator would reduce the collision rate and save more power. Therefore, it would be beneficial to employ both MAC schemes for the application and switch between them depending on the scenario.



The MAC Layer Architecture (MLA) [KHCL07] allows the user to easily select a MAC algorithm in the make file during compile time. Beside B-MAC, X-MAC and SCP, it provides TinyOS’ BoX-MAC and a pure TDMA protocol (called “PureTDMA” in the following), which — according to the publication — is similar to the GTS portion of the IEEE 802.15.4 standard. Internally, all MAC algorithms contain a top configuration with the same name, and the requested algorithm is included by adding the right directory to nesC’s include path. To include more than one of these MAC algorithms, these top configurations simply need to be renamed uniquely. Since they already expose the same interface no API unification as in Section 5.2.1 is needed. Each algorithm can also provide algorithm specific interfaces by the `MacControlC` configuration. For the PureTDMA algorithm, `MacControlC` can be ignored; for BoX-MAC, we directly wire a component that sets the LPL wakeup interval to a default value. Moreover, we corrected some minor errors and completed functionality such as the missing `Send.cancel` command for BoX-MAC. Apart from that, we keep our modifications minimal although, for example, switching the radio off and on again could be avoided when switching algorithms.

Multiplexing Layer

new SLOC	671
new files	3

Isolation Layer

	BoX-MAC	PureTDMA
new SLOC	255	141
new files	10	6
changed SLOC	4	3
changed files	3	1

Table 6.2: Overview generated code for MAC algorithm switch

Table 6.2 gives an overview on the amount of generated and changed code by Tiny-Switch. As for the sample application in Section 5.2, “SLOC” denotes the source lines of code, i.e. the source code lines without comments and blank lines. Compared to the multiplexing and isolation layer for the routing algorithm switch (Table 5.1) the new SLOC is double as high due to the high number of interfaces that need to be multiplexed and isolated for the MAC algorithms that are deeper embedded in the network stack.

Our application consists of a base station and a node part. The base station application regularly sends beacons and simply stores the data received from the other nodes. When these nodes have received a certain number of beacons, they assume to be in the vicinity of the base station and start sending data. We have built three different application pairs that behave differently in this case: the non-adaptive application

continues to use LPL; the application with parameter-based adaptation effectively disables LPL by setting its intervals to 0; and the TinySwitch-based application switches the MAC protocol to TDMA. Note that this switch is not controlled by TinyAdapt but by the application layer, which detects the presence of the base station. Both adaptive applications return to normal operation when they have transmitted the stored data.

We performed our experiments in the TWIST testbed. Since mobility is not supported by TWIST, we simulated the appearance of the base station by simply switching it on and off. 19 nodes in the vicinity of the base station node formed the “horse nodes”, and the amount of data to be sent to the base station was 60000 bytes.

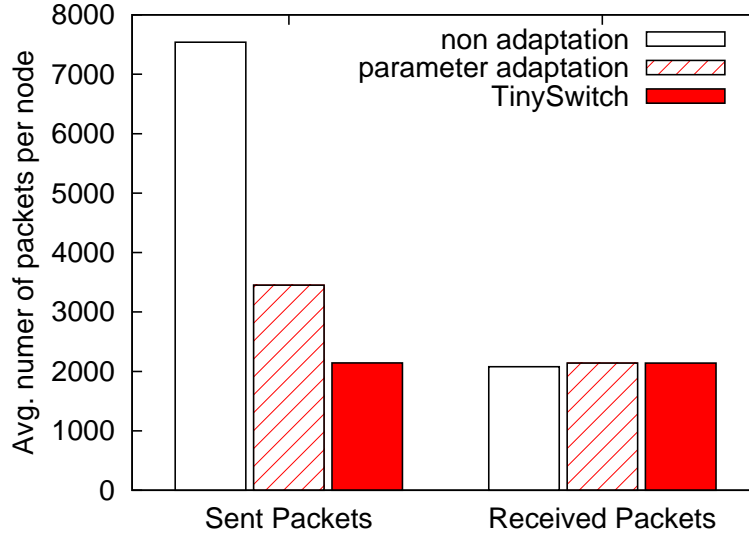


Figure 6.14: Number of received packets in horse monitoring scenario for applications performing no adaptation, parameter-based adaptation and using TinySwitch

Figure 6.14 shows the average number of data packets sent and received in 5 testbed runs. All applications succeed in delivering almost all packets to the base station without significant difference. However, in the non-adaptive application, this is achieved by LPL repeating every packet 3.5 times on average until an ACK is received from the base station. The second application without LPL sends a data packet only once, but will resend multiple times when it does not get an ACK from the base station, which causes a 61% packet overhead. Finally, the TinySwitch application uses TDMA to send the data and, therefore, almost no collisions occur — only with nodes that have not detected the base station yet.

The different number of sent packets also affects the energy consumption of the network (excluding the base station). Since energy cannot be measured in TWIST the three applications are simulated with Avrora, which achieves similar packet numbers as TWIST and, therefore, is comparable. Fig. 6.15 shows these simulation results. Energy consumption increases considerably for the applications that continue to use BoX-MAC

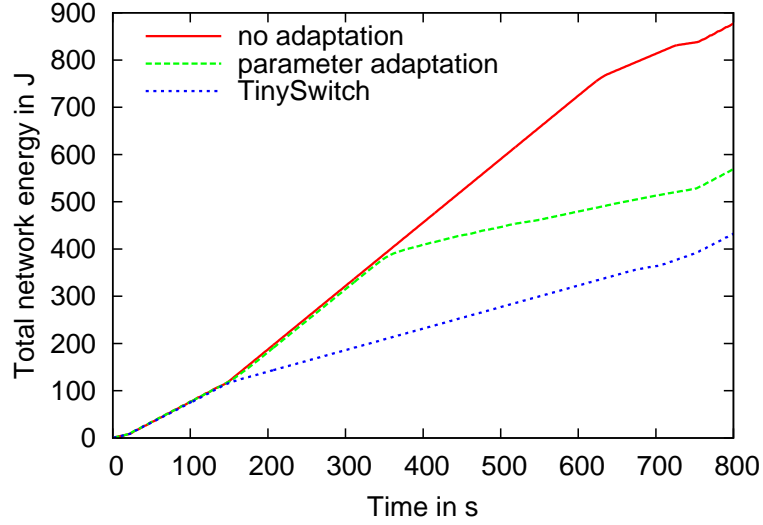


Figure 6.15: Network energy consumption in horse monitoring scenario for applications performing no adaptation, parameter-based adaptation and using TinySwitch

when they start to send data (at  $\approx 150s$ ). Since the channel is almost never free, LPL hardly switches off radio in the non-adaptive version and, therefore, there is no difference at the beginning to the parameter-based adaptation. The latter finishes first (at  $\approx 350s$ ) sending the data and turns on LPL again, effectively reducing its energy consumption. In contrast, the use of TDMA in the TinySwitch application decreases needed energy, and although it takes longer to send all the data (until  $\approx 670s$ ) its overall performance is best. The switching process takes only  $\approx 2.3ms$ , which is dominated by switching off and again on the radio, and, therefore, enables fast change of operation mode.

In summary, this section shows that with minimal effort TinySwitch can make MAC algorithms switchable that were originally not designed for this purpose. In the presented scenario, their use results in significant increased performance. It also confirms that the concept presented in Chapter 5 is feasible at least as standalone solution. In the next section it will finally be combined with TinyAdapt.

## 6.6 Case Study: Switching of Routing Algorithms

Our third case study focuses on autonomous adaptation by switching of routing algorithms due to network dynamics. It shows the complete TinyAdapt framework as presented in Chapters 3 and 4 including the mobility metric and monitoring system as well as TinySwitch as presented in Chapter 5. The case study has two parts: the first part is based on Avrora 1.7.115 and evaluates adaptation due to node mobility, the

second part is performed in the TWIST testbed and shows adaptation due to node interference.

In our case study application, a sensor network periodically sends monitoring data to a base station. It uses TinyOS' Collection Tree Protocol (CTP) to do so, which works well as long as the radio links are stable. However, in case the nodes start to move (e.g. because the objects they are attached to move or are moved) or communication is disturbed (e.g. due to electrical interference) no longer-term stable links can be found any more. Although the tree-based algorithm might be exploited to rebuild the tree more often to accommodate to the unstable environment an algorithm tailored to the mobility pattern or, as used in this case study, even a simple flooding algorithm suits better the scenario.

To achieve this, we created an application that can switch between both algorithms. In Section 5.2, we have already shown how to integrate CTP and Flooding into a single application using TinySwitch. This switchable routing stack is used here. The control interface of TinySwitch was connected to TinyAdapt in order to let TinyAdapt autonomously decide between stable and unstable network conditions and to select the appropriate algorithm. Additionally, we allowed several algorithm parameters to be controlled and adapted by TinyAdapt through TinyXXL: for CTP, the maximum beacon interval and the maximum number of retries when sending a data packet from node to node as well as the wakeup interval for Low Power Listening (similar to the first case study in Section 6.4); for Flooding, the maximum number of stored broadcast sequence numbers for duplicate detection, the maximum jitter when sending a message, and the number of retransmissions for each packet.

### 6.6.1 Adaptation due to Mobility

In the experiment scenario, we consider a logistics application in which containers are moved around. Nodes attached to these containers send their location and other measurements to a base station. The experiment is divided into three sub-scenarios, a *static*, *mobile* and *dynamic* one. In the static scenario, we arrange 25 MicaZ nodes that are 14m apart as a 5x5 grid in the middle of a 100x100m<sup>2</sup> area. Initially, nodes boot randomly and, after 5s, they send a message every 10s to the base station. The dynamic scenario extends the static one. Namely, after 180s, we instrumented Avrora to start moving the nodes (except for the base station) with a speed of 1m/s and no wait time. After 600s, the grid position from the beginning of the simulation is set as new destination point. After some time, the static grid scenario is re-built and the simulation continues until 1200s. The mobile scenario consists only of the mobile phase, i.e. all nodes except for the base station are moving all the time.

We simulated different combinations of the algorithm parameters with the node speeds 0, 1, 2, and 4m/s, and measured the needed power, delivery ratio and latency. Using the post-processing method in Section 4.6 we finally obtained 28 combinations under the assumption that higher delivery ratio and lower power consumption and latency are

## 6.6 Case Study: Switching of Routing Algorithms

preferred and that the user will choose values with a granularity of 5 percentage points for the delivery ratio,  $3mW$  for the power consumption and  $0.5s$  for the latency.

The goal definition as given in Table 6.3 is used to control the adaptation process and as a guideline when manually selecting parameterisations for reference applications in the following.

Requirements	1. delivery ratio $\geq 90\%$ 2. end-to-end latency $\leq 5s$
Relaxation	decrease requirement 1 in steps of 20 until 50
Optimisation	minimise power consumption

Table 6.3: Goal Definition for Logistics Application

We built four applications: a non-adaptive version that uses CTP and the best parameter set for static nodes ( $N-S$ ), a non-adaptive version that uses Flooding and the best parameter set when nodes are moving at  $1m/s$  ( $N-M$ ), and two adaptive versions that include TinyAdapt, one ( $A-S$ ) that starts with the same configuration as  $N-S$  and one ( $A-M$ ) that is preconfigured to initially use the same settings as  $N-M$ .

	N-S	N-M	A-S	A-M
static scenario (mW)	10.9	50.6	16.3	—
mobile mobile (mW)	—	25.5	—	29.2

Table 6.4: Comparison of power consumption without ( $N-S$ ,  $N-M$ ) and with TinyAdapt ( $A-S$ ,  $A-M$ ) in different scenarios

We ran 10 simulations with different application version-scenario combinations to determine the overhead of the adaptation. When comparing version  $N-S$  with  $A-S$  and  $N-M$  with  $A-M$  (see Table 6.4) the higher power consumption of the adaptive versions reflects the additional beacon or metric packets that need to be sent by TinyAdapt. Since the Flooding algorithm causes more routing packets in the network, the overhead decreases due to more piggybacking. The  $N-M$  version works both in the static and mobile scenario. However, it consumes a multiple of the power consumed by  $N-S$  or  $A-S$ . All versions have similar results regarding the packet delivery ratio: in the static scenario 97–99% of all packets are delivered and 58% in the mobile scenario. We did not simulate  $A-M$  in a static scenario and  $A-S$  in a mobile scenario since it will adapt quickly, so no comparison can be made. In addition, we did not use  $N-S$  in the mobile scenario since CTP will deliver almost no packets.

The benefits of TinyAdapt become obvious when we run version  $A-S$  in the *dynamic* scenario, which starts with the static part and the mobile part follows. (Note that  $A-M$  would work in the same way, but it first needs to adapt to the static network at the beginning.) Figure 6.16 depicts the packet delivery ratios within a  $1s$  interval.

During the first  $180s$ , the delivery ratio is 99% on average as predicted by the pre-installation simulations, but when the nodes begin to move (line  $A$ ) practically no

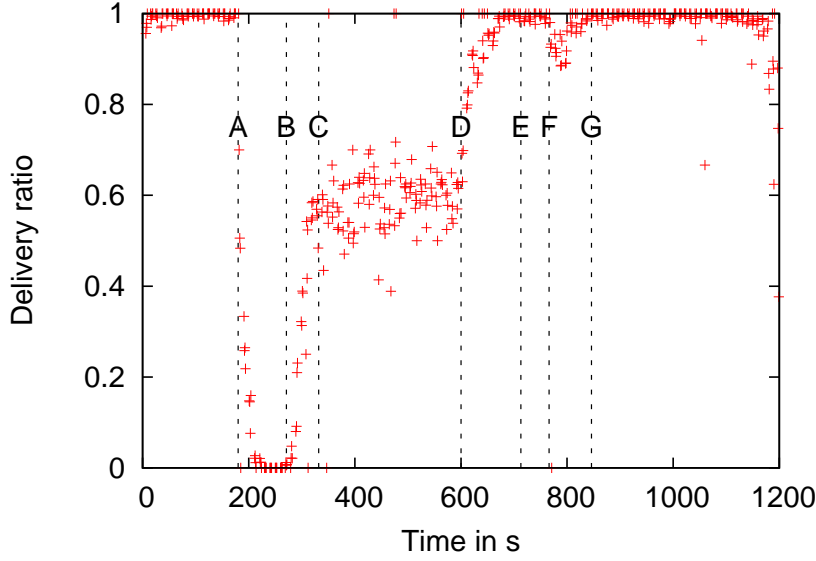


Figure 6.16: Delivery ratio in the dynamic scenario with TinyAdapt

packets are delivered. The first nodes start to adapt at time 271s (line *B*) by switching to Flooding and at this point the delivery ratio starts to increase again. The delay between the start of the movement and the start of the adaptation is caused by the exponential moving average and a duration of 30s in which the metric has to stabilise until the adaptation is triggered. At time 332s (line *C*) all nodes have adapted. On average, the delivery ratio stabilises at 58%, which is consistent with the simulation of the pre-installation phase. All nodes start to move towards their original grid position at time 600s (line *D*). Since the network becomes more and more connected the delivery ratio increases. When the movement comes to an end at 713s (line *E*) 98% of all packets are delivered again. The first nodes start to adapt at 766s (line *F*) and all nodes have finally adapted and switched back to CTP at 846s (line *G*). During the switch, the delivery ratio drops for a short time, but recovers quickly to 99% as at the beginning of the simulation.

In Figure 6.17, the development of the overall required network energy is plotted for the same simulations. In the static zones, the curve increases slowly since CTP uses the least power in static scenarios. When nodes started to move but before the nodes started to adapt (between lines *A* and *B*) the slope is quite high since CTP resends each packet very often although the parent is not available any more. Flooding requires more power than CTP as can be seen by comparing the slope between lines *C* and *D* with the slope from the beginning or the end. When the network becomes static again (after line *D*), the slope becomes steeper again since a message will reach more and more nodes by Flooding, thus wasting energy. The adaptation back to CTP is indicated by a drop of the slope between lines *F* and *G*.

The non-adaptive application using CTP (*N-S*) consumes the least power, but it will deliver almost no packets in mobile environments while increasing the consumed power.

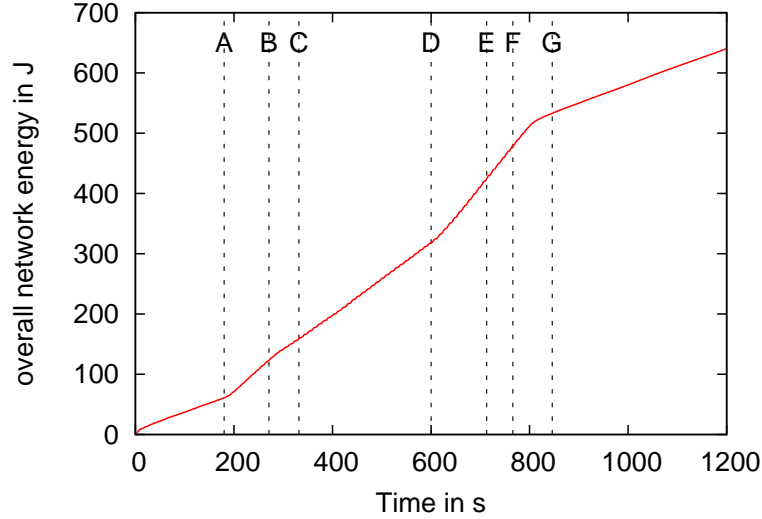


Figure 6.17: Energy requirement in the dynamic scenario with TinyAdapt

The non-adaptive application using Flooding ( $N-M$ ) works well in static and mobile environments with respect to the packet delivery ratio. However, it requires significantly more power in static scenarios. The application with TinyAdapt ( $A-S$ ) incurs some overhead to constantly monitor the network, but the application always delivers the same amount of packets as that of the non-adaptive versions in the corresponding parts of the dynamic scenario.

### 6.6.2 Adaptation due to Interference

The following tests have been performed on 96 nodes of the TWIST testbed. Since node location information is unavailable, we first mapped the network of 96 available nodes using a transmit power of 3 dBm. Then, we selected one node as the sink and six other nodes at the other end of the network as data sources, sending data every 10 seconds to the sink. We also placed 4 interferer nodes “in the middle” of the network. On all but the interferer nodes, we installed our test application and logged the packets sent from the data sources and the packets received at the sink for 30 minutes: after the first 10 minutes, we activated the interferer nodes, which sent out interfering signals [BHL<sup>+</sup>09] for 8–12 seconds and then paused for 8–12 seconds; after another 10 minutes, we deactivated the interferers and observed the network for the final 10 minutes.

Figure 6.18 shows the delivery ratios calculated over 30s intervals of four testbed runs using pure CTP and Flooding, respectively. At the beginning, both algorithms managed to transmit all packets to the sink. When the interferers were activated (point A) the delivery ratio immediately started to decrease for CTP and after 80 seconds only some packets arrived at the sink. For Flooding, the delivery ratio started to vary widely since Flooding tries every possible path to the sink for data delivery, which

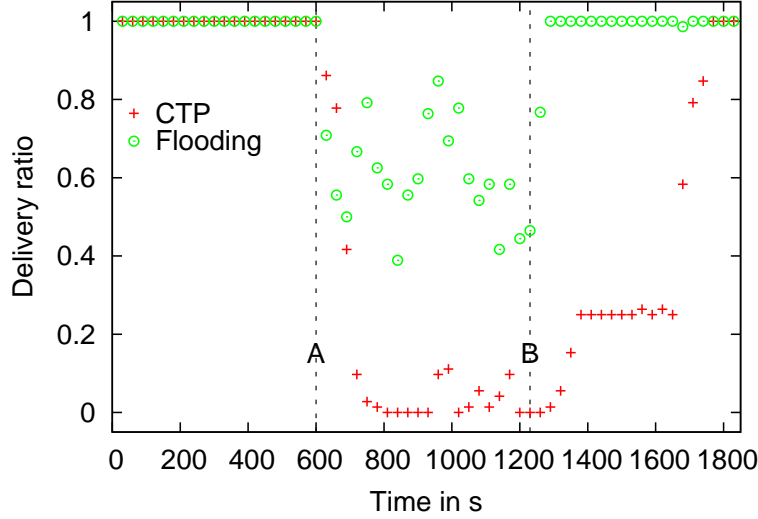


Figure 6.18: Delivery ratios for pure CTP/Flooding applications with interferers between time A and B

was successful in 57% of the packets on average. After the interferers were deactivated (point B), the ratio for Flooding almost immediately returned to 100% while it took variable times for different runs until CTP could recover and found a path to the root node again. For one run, it took approx. 2 minutes while the others recovered after 7.5-8.5 minutes.

While Flooding seems to outperform CTP, it requires more resources since the messages are distributed in the whole networks, even in areas with no route to the sink. Moreover, Flooding results in higher probability of collisions when all nodes are trying to rebroadcast packets. Therefore, under stable conditions CTP should be used, but during unstable periods Flooding should be employed. To achieve this, we again used the adaptive application that can switch between both algorithms using TinySwitch. The difference is that there are only two “velocity” classes and four entries in the configuration table, containing the performance of CTP and Flooding with their default parameters in both scenarios.

Figure 6.19 shows the delivery ratios of this adaptive application. We can observe that the delivery ratio still decreases dramatically for CTP after the interferers were activated (at point A). After 140 seconds, TinyAdapt detected the poor performance and decided on a switch to Flooding using TinySwitch (during the interval indicated by C). Immediately, we can see that the delivery ratio increased to an average of 54%. After the interferers were turned off, it took another 140 seconds until TinyAdapt decides to switch back to CTP. This happens during the interval indicated by D. We can see that some messages were lost since there is no soft transition between the two algorithms. Note that intervals C and D show when the first nodes starts and the last nodes ends with adaptation. The actual adaptation at a single node takes only  $\approx 2$ ms in C and 0.9ms in D, including making the adaptation decision.



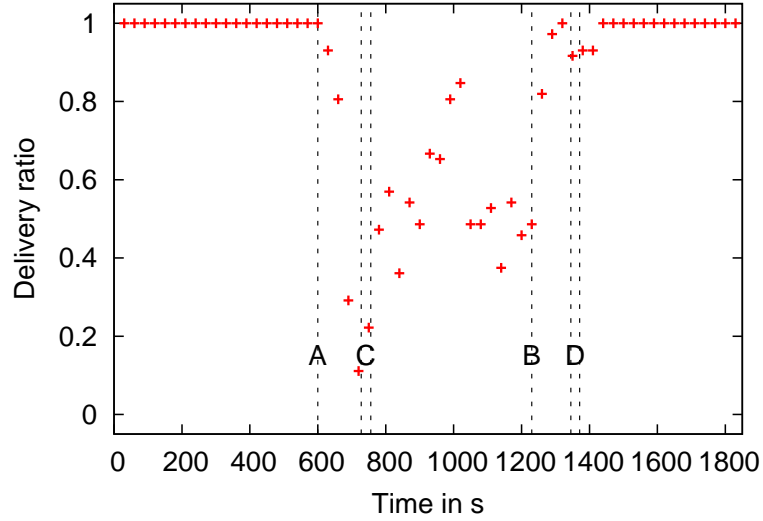


Figure 6.19: Delivery ratios for adaptive application using TinyAdapt and TinySwitch with interferers between time A and B and adaptation at C and D

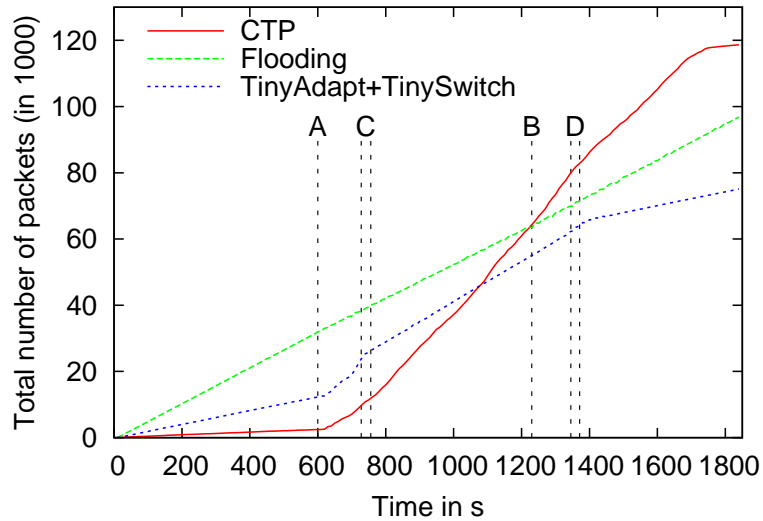


Figure 6.20: Total number of packets sent by different routing applications

Compared to the CTP version, the overall delivery ratio of the adaptive version is much better; but compared to the Flooding version, the adaptive version seems to have no advantages. However, with pure Flooding each packet would be rebroadcasted by all nodes in the network, resulting in heavy network traffic (see Fig. 6.20). In contrast, CTP performs very well during the first 10 minutes, reducing the number of packets by 92%. This changes drastically when the interferers are turned on at point A. CTP tries to find new routes and resends each packet up to 30 times if it cannot be delivered, resulting in a high packet count. Only after CTP recovered at the very end of the measurement the initial behaviour is restored. After the interferers have been turned on (point A), the adaptive version still behaves as CTP and the packet count

increases heavily. After adaptation in interval C it immediately runs as Flooding until in interval D original behaviour is restored. The monitoring packet overhead of the adaptive version can be seen when comparing CTP and the adaptive version during the first 600s. However, already short but recurring periods of interference outweigh this overhead. Moreover, the monitoring beacons are still sent every 5 seconds, which could be reduced since no distinction of different velocities is needed.

In summary, this case study demonstrates for both scenarios that the adaptive application using TinyAdapt and TinySwitch outperforms the other two applications and results in higher overall delivery ratio while being resource efficient.

Overall, in scenarios with network dynamics no version of the example routing algorithms is able to achieve a constantly high delivery ratio and low power consumption. However, TinyAdapt can detect the change and manages to adapt to this dynamic by switching the algorithms using TinySwitch and, therefore, combines both worlds in a unique way. This also shows a successful implementation of the TinyCubus concept that was presented in Section 2.2.

## 6.7 Overhead Analysis

Although for both TinyAdapt and TinySwitch much work is done during the pre-installation phase, which is supported by applications running on the development PCs, both frameworks include code that is to be executed during run-time on the sensor node. In this section, this overhead is analysed with respect to memory usage and processing cost.

Since TinySwitch is typically included in adaptive applications using TinyAdapt, we first analyse TinySwitch alone and then a complete application, which includes TinyAdapt, TinyXXL and TinySwitch.

### 6.7.1 TinySwitch

As explained in Section 5.2.3, a `switch` statement is added at source code level in the multiplexer for each function that can be called in the API, i.e. each command of a provided interface and each event of an used interface. Similarly, a simple `if` statement is added for each function that is called from an API component, i.e. each command of an used interface and each event of a provided interface, and for all commands and events of the interfaces to be isolated in the isolation layer.

As examples, we examine the applications in our second and third case study (Sections 6.5 and 6.6). They were built with TinyOS 2.1.2 using `msp-gcc` 4.6.3 and `avr-gcc` 4.1.2, for TelosB and MicaZ, respectively. In both applications, the switching layer selects one of two algorithms. The machine code for an isolated `switch` statement needs 18 bytes on TelosB and 22 bytes on MicaZ; the simple `if` statement requires 6 bytes on

		Case Study 2	Case Study 3
Multiplexer	provided commands & used events	17	6
	used commands & provided events	21	2
	TelosB code size increase	700	352
	MicaZ code size increase	1094	326
Isolation	commands/events	42	10
	TelosB code size increase	392	72
	MicaZ code size increase	852	252

Table 6.5: Analysis of multiplexing and isolation layer for case studies 2 and 3

TelosB and 10 bytes on MicaZ. The compiler for MicaZ adds an unnecessary boolean conversion for the `if` statements in the isolation layer and, therefore, they require 16 bytes there.

Table 6.5 shows the number of provided/used commands/events and the increase in code size for both platforms in the applications. Note that the generated code of the multiplexing layer also contains the TinySwitch management code. It is obvious that in general the additional code mostly depends on the number of interfaces to be multiplexed/isolated and the number of their commands and events. Thus, algorithms with many dependencies such as the algorithms of the MAC Layer Architecture [KHCL07], which are deeply embedded into the network stack, lead to bigger TinySwitch code. However, due to heavy inlining and other optimisation decisions by the compiler no general number for the increase of the binary's size can be given. But even for extreme cases like the MAC Layer Architecture the code size is  $\approx 2\text{kB}$  for MicaZ and  $1\text{kB}$  for TelosB. For algorithms that provide only a few interfaces and only use some external modules such as the routing algorithms in our second case study, the code size increase stays well below  $1\text{kB}$ . Thus, TinySwitch needs much less code memory than the common reprogramming technique Deluge (see Sect. 6.7.3). Moreover, concerning RAM overhead, TinySwitch only introduces 5 byte-size variables that allocate between 6 and 10 bytes in our applications, depending on the compiler adding fill bytes for data alignment.

During run-time, execution of the `switch` and `if` statements is quite fast: In our applications with two switchable algorithms the maximum additional delay in a `switch` statement is 20 cycles ( $5\mu\text{s}$ ) for TelosB and 16 cycles ( $\approx 2.2\mu\text{s}$ ) for MicaZ. The `if` statements are even faster and take only 7 cycles ( $\approx 1.8\mu\text{s}$ ) on TelosB and 5 cycles ( $\approx 0.7\mu\text{s}$ ) on MicaZ. However, in the isolation layer the unnecessary code for MicaZ results in an additional 8 cycle ( $\approx 1.1\mu\text{s}$ ) delay. Compared to typical functionality that is performed, for example, by the implementations of the Send/Receive interface commands/events, this overhead is much smaller and, thus, negligible. Especially the isolation layer works very fast so that the delay to possibly time critical events is not noticeable.

### 6.7.2 TinyAdapt

At run-time, TinyAdapt includes several components as shown in Figure 3.2. We measure the increase of the code size and RAM usage for the logistics application used in the third case study. As baseline, we created an application that includes both CTP and Flooding and the TinySwitch components to switch between them. Then, we added one TinyAdapt component after the other. Table 6.6 shows the additional ROM and RAM sizes for each component as well as their sum and the total size for the application.

	TelosB		MicaZ	
	Code	Data	Code	Data
General Piggybacking	1608	84	5658	77
Dissemination	1302	148	2274	141
Monitoring	5000	180	5020	145
Adaptation	1496	270	1780	268
Total Increase	9406	682	14732	631
Total Application Size	30302	3010	40844	2882

Table 6.6: TinyAdapt (module) memory usage and complete size for Case Study 3

In general, code size increases more for MicaZ. When the general piggybacking module is included, the code size increases much more for MicaZ than for TelosB. The current piggybacking implementation uses 32 bit time stamps directly from TinyOS, which require more code on the 8 bit Atmega128 (MicaZ) than on the 16 bit MSP430 (TelosB). Additionally, a time comparing function is inlined in several functions, creating a lot of code to re-calculate its parameters and to compare the values. We do not know why this inlining happens here since it leads to an quite significant code size increase but the compiler was instructed to minimise code size (compiler switch `-Os`).

Note that the Monitoring component also includes the mobility metric. Here, the mathematical calculations include floating point operations, which require additional library functions. For example, on MicaZ they require more than 1000 bytes.

On the other hand, the TelosB binary requires more RAM. This is due to necessary memory alignment since the 16 bit MSP430 requires non-byte variables to start at an even memory address, which sometimes requires additional padding bytes. The additional RAM for the adaptation component is mostly allocated by the Configuration Table, which uses 224 bytes in this application. As explained, this depends on the table reduction process.

Looking at the total application size, the approach with TinyAdapt and TinySwitch seems feasible and more application code or additional algorithms could be added. On TelosB, approx. 12kB of program memory are free. The first limit to be reached is probably the RAM size of MicaZ. Although approx. 1kB of RAM seems to be free, this needs to be shared with the stack. Currently, the CTP components use 1113 bytes

of RAM, while the Flooding components use 690 bytes. The final adaptive version, which we have examined here, includes both algorithms without sharing any variables. However, message pools could be shared by moving them to TinyXXL and, thus, the needed RAM could be reduced by at least 500 bytes.

The computation of the exponential moving average takes  $0.46ms$  on MicaZ. Despite the use of emulated floating point arithmetic, this is quite short and can, therefore, be executed on the fly to update the final metric once in every beacon interval of  $5s$ . The execution time of the adaptation process depends on the the size of the Configuration Table and the difficulty to find a matching entry.

In the logistics application of the third case study, the Configuration Table consists of 28 entries. For the static phase, only restrictions and optimisations of Table 6.3 have to be applied which takes  $0.7ms$ . When the nodes start to move, the maximum possible delivery ratio is 66% according to the Configuration Table. To select this entry, two relaxation steps with subsequent re-evaluation of the restrictions are necessary. Thus, the execution time of the adaptation process increases to  $1.8ms$ . This is executed in a TinyOS task to allow for parallel execution of events. Nevertheless, it is quite short and in a typical monitoring application with a lot of idle time all presented computations can be performed easily without affecting the actual application.

In total, analysis of the run-time overhead confirms the efficiency of the TinyAdapt and TinySwitch solutions. Therefore, the frameworks are suitable for standard sensor node hardware as required by the *Generic Applicability* principle of Section 3.2.

### 6.7.3 Comparison to Node Reprogramming

The main advantage of TinySwitch is to avoid the need to reprogram the node for algorithm switching (see Section 2.3.2 and Chapter 5). In this section we analyse the overhead of TinyOS' standard reprogramming solution Deluge [HC04]. In Deluge, for each alternative algorithm a complete application binary has to be built and transferred to the serial flash of the mote using Deluge's base station functionality. Then, reprogramming of the node can be initiated by the **NetProg** interface. Deluge will reboot the node and reprogram it during boot.

To analyse the reprogramming process in detail we used a measurement setup as shown in Figure 6.21. The sensor node is powered by a stabilised 3V power supply, and in the anode line we put a  $100m\Omega$  resistor. Given a maximum current drain by the sensor node of  $30mA$  the voltage drop at the resistor will be  $3mV$  at max, which does not affect the node. Nevertheless, the voltage drop is proportional to the current drain, which can be measured using the data acquisition module NI USB 6212 (connection B). Also, the real voltage is measured (connection A) and, thus, the power and energy can be calculated. To get the information, which phase of the reprogramming process is currently executed, the GPIO2 and GPIO3 pins of the TelosB motes or the PW0 and PW1 pins of the MicaZ motes were connected to two further ports (connection

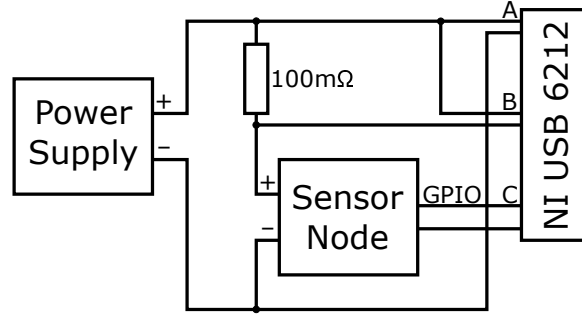


Figure 6.21: Setup for power measurement of sensor nodes

C). We used a sampling rate of 100kS/s and smoothed the gathered values for the current drain afterwards by averaging 1000 samples.

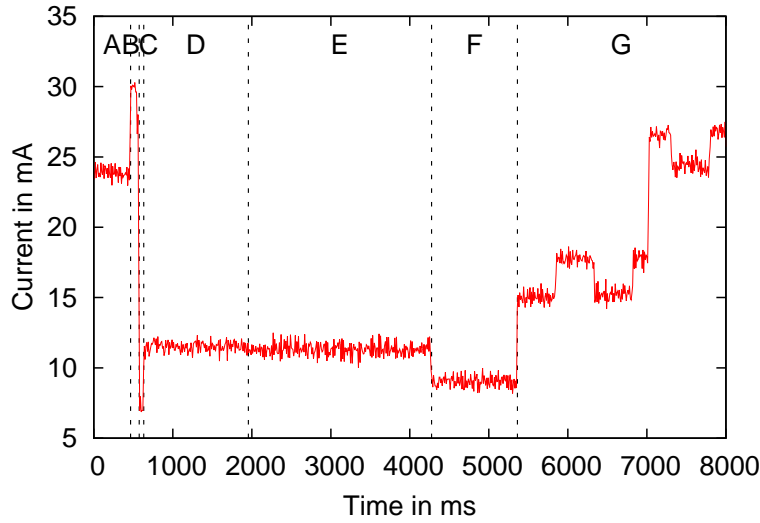


Figure 6.22: Current drain of MicaZ node during reprogramming

Figure 6.22 shows the current drain of the complete node when reprogramming a MicaZ node using Deluge. In phase A the application is running normally with the radio turned on. Then, it initiates reprogramming in phase B using Deluge’s **NetProg** interface. This restarts the node, which can be seen as a sudden current drop. The current remains at 7.2mA in phase C when the bootloader (also delivered with TinyOS) is initialising. In phase D, the bootloader reads the new binary completely from the serial flash and verifies it, which leads to an increased current of 11.5mA. This step takes 1.3s for this 29.5kB binary. When the verification succeeds, the binary is read again from serial flash and the MCU’s on-chip flash memory is reprogrammed in phase E, which takes another 2.3s. The current remains almost stable at 11.3mA. Finally, the new binary is started. At startup, the clock is calibrated in phase F. During this, the serial flash is off again and, thus, the current drops to 9.2mA only. Afterwards, the main application is running in phase G, blinking the red LED, which can be recognised in the current drain. In the background, Deluge reads the complete flash again and

verifies the provided images, and finally the radio is started, which results in the final current increase. In total, approx. 40.2mJ are needed per 10kB binary when taking into account only the pure verification and reprogramming phases (D and E).

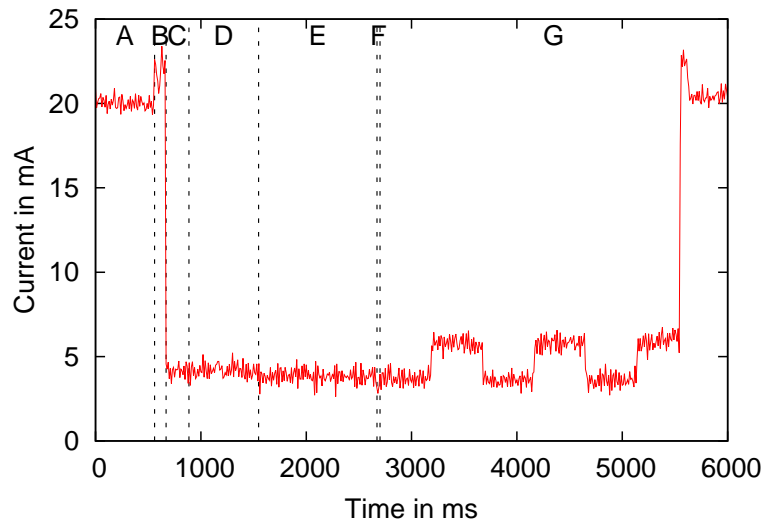


Figure 6.23: Current drain of TelosB node during reprogramming

For the TelosB nodes, similar current drains hold for reading the serial flash and for reprogramming the MCU (see Figure 6.23), but only 0.7s are needed for a 25kB binary to read from serial flash and 1.2s to reprogram, thus summing up to 9mJ per 10kB binary. Note that phase C is a little bit longer on TelosB since the digitally controlled oscillator is calibrated here. Instead, no calibration is done in phase F, thus shortening in total the startup time.

Although the required energy for reprogramming seem small for both platforms, avoiding reprogramming allows the possibility to also obtain small power savings by switching to a different algorithm. Moreover, depending on the motes, the size of the alternative binaries and the adaptation frequency, power saving can be considerable.

To use the reprogramming function of the `NetProg` interface Deluge needs to be included in the application binary and also the bootloader needs to be installed on the node, both consuming memory. Deluge requires 13186 bytes code and 734 bytes data on TelosB and 16500 bytes code and 596 bytes data on MicaZ when including it in the `RadioCountToLeds` application. The bootloader adds another 1898 bytes of code on TelosB and 2162 bytes on MicaZ. Of course, if Deluge is used to provide over-the-air reprogramming facilities for error correction there is no extra memory overhead for application-initiated reprogramming.

Finally, it should be noted that reprogramming is only possible when the supply voltage is 2.7V minimum. Thus, it is possible that a node gets stuck in a binary that consumes more power than an alternative one and cannot be reprogrammed due to low battery voltage. Therefore, reprogramming is not suitable for situations when an “emergency routine” should be run due to low voltage to keep at least minimal operation. With

switching, all parts of the code that are not needed for this basic functionality can simply be replaced by dummies, thus reducing power consumption considerably.

### 6.8 Conclusion

The evaluation shows that TinyAdapt and TinySwitch fulfil the main design principles established in Section 3.2: Ease of Use, Generic Applicability, and Effectiveness. For this reason, they outperform other adaptation and code exchange solutions that lack in at least one of these principles.

All necessary parts to create an adaptive application are contained in the TinyAdapt framework. If adaptation is to be performed by parameterisation, only a few and easy modifications to use TinyXXL are necessary. The decoupling of the open framework and the algorithms led to a generic solution that is not restricted to a certain type of algorithm and supports different methods of adaptation.

We showed that the framework can run on standard sensor nodes with low overhead. The autonomous adaptation process is controlled by Goal Definitions that express user preferences in an easy and comprehensible way. In our experiments, TinyAdapt managed to meet these goals all the time without exploring the design space during run-time, but based on previous simulations, making the adaptation decisions very efficient.

The evaluation of TinySwitch highlights its flexibility and efficiency in application performance enhancement. We showed that TinySwitch can be invoked directly by the application, or it can be easily integrated with an adaptation framework and is transparent to the application. Also, TinySwitch can be used at different layers. In all cases, TinySwitch improves the performance compared to a non-adaptive application. Moreover, TinySwitch is lightweight, adding only minimal run-time overhead, especially compared to node reprogramming, which requires a considerable time and amount of energy.

Therefore, TinySwitch is perfectly suiting TinyAdapt since it can efficiently activate the appropriate algorithms without reprogramming of the node in reaction to dynamic network changes.



# 7

## Summary and Outlook

The complex nature of today's Wireless Sensor Networks require more intelligent management systems. There are no single adjustable screws to easily tune the whole network. Instead, normal end-users without deeper knowledge on the inner workings of the sensor network will commonly use them in the future.

However, real-world installations do not work in stable, controlled and reliable environments but experience the rough reality of the “world outside”. The users of a sensor network do not want to adjust it manually — if they have the capability to do so at all. On the other hand, these users will change their requirements towards the sensor network over time since working with big data delivered by sensor networks is often explorative and new questions arise during their analysis. For these reasons, sensor networks need to have adaptive capabilities. They need to support both the dynamic environments and the changing requirements of the users.

In this thesis, we have developed the generic adaptation framework TinyAdapt that solves this major challenge and allows for the easy creation of efficient adaptive WSN applications. To achieve deterministic adaptation, possible algorithms and their parameterisations are tested in a pre-installation phase and only the best combinations are included in the final application. During run-time, the user can explicitly and easily define the performance requirements for the application, and TinyAdapt will select from the tested combinations and reconfigure the network. A detailed workflow guides the different actors through the process. TinyAdapt provides different applications to configure the adaptation, evaluate the different configuration possibilities and select the combinations to be installed in the final application. In doing so, TinyAdapt is not restricted to specific algorithm classes, their parameters, network or performance metrics, test methods or even operating systems. We have implemented TinyAdapt for TinyOS to show its feasibility.

Beside reconfiguring running algorithms, it is often necessary to exchange the algorithms completely to fulfil the adaptation goals. Since this is difficult during run-time in TinyOS we have developed TinySwitch, a framework to switch between alternative algorithms that are installed in parallel. While switching has been proposed earlier, TinySwitch is the first solution that creates all switching code for arbitrary algorithm classes automatically. Especially in TinyOS, it is difficult to discover all external dependencies of an algorithm that is usually distributed over different components and

then to switch or isolate all of them. TinySwitch carries out this tedious task and involves the programmer only when necessary.

The evaluation of TinyAdapt and TinySwitch, done by simulation and in testbed experiments, show the benefits for real-world applications. Firstly, changing requirements of the user can be met even for very different optimisation goals. Secondly, different application states can be supported by a changing MAC layer implementation. And thirdly, applications with different real-world operating conditions, e.g. mobile and static phases or unstable communication environments, achieve optimal data delivery with our frameworks while being energy efficient.

In conclusion, the presented frameworks TinyAdapt and TinySwitch perfectly support the TinyCubus vision. They integrate with available modules of TinyCubus and extend it with adaptation and code switching capabilities. Altogether, the vision of flexible, self-configurable, self-adaptive, and efficient Wireless Sensor Networks seems to come true now.

### 7.1 Future Work

There are several possible research directions that could extend the work of this thesis.

TinyAdapt considers a complete sensor network, but conditions can prevail only in smaller parts of a large network so that local adaptation could improve the application's performance. The monitoring component could detect such regions and determine its borders. Contour maps [MNLL06] or eScan [ZGE02] could be employed here. However, different algorithm parameterisations or even different algorithms are often incompatible, e.g. because they communicate using different timings or message structures. Therefore, gateway nodes need to be defined on which several configurations are active. Although for unicast packets a multiplexing layer could automatically decide which algorithms to use, establishing general methods is difficult since some communication algorithms are inherently incompatible, handling of broadcast packets requires special attention, and other distinguishing features need to be found for other types of algorithms.

Novel flooding techniques such as Glossy [FZTS11] can be examined to improve the monitoring system because they are fast and energy efficient. However, since a single node needs to initiate the flood selection, mechanisms for this initiator have to be established that are robust also in case of temporary network partitions.

The monitoring system and the proposed mobility metric could be adaptive itself. In case of low variance of the calculated metrics the frequency for gathering the network-wide values can be reduced without affecting the adaptation quality or speed. In general, only nodes with high variance close to the current maximum can request a higher exchange rate. In the same way, the amount of beacons can be reduced for low

speeds since the chance of missing neighbours is reduced then. Since the detection of higher speeds is worse with lower beacon rate, the triggers to alter the beacon rate and the influence on adaptation quality and speed need to be examined carefully.

Also, the calculation of mobility metric could be enhanced. Currently, it makes extensive use of floating point arithmetic. Although the overhead concerning code size and execution time is acceptable, the use of more efficient calculation seems to be feasible, e.g. fixed point arithmetic.

In TinySwitch, alternative algorithms must not share internal components that are not generic. Currently, the programmer needs to change these components to generic components manually, which can be automated. As optimisation, the programmer could mark components as “stateless” to allow such usage. Alternatively, the components can be explicitly unblocked in order to share their state between different algorithms or they can be reinitialised by the framework.

Data of alternative algorithms is currently kept separate, which increases RAM consumption. Since only one algorithm is enabled at a time, RAM of alternative algorithms could be shared. For native support, the C file created by nesC could be modified in such a way that all data that belongs to an algorithm is kept in a `struct`, and structs from alternative algorithms are encapsulated in a single `enum`. Of course, all accesses to this data need to be rewritten. Also, the programmer needs to make sure that all variables are initialised correctly when the algorithm is started.

To follow the original vision of TinyCubus of having multiple applications running on top of it (see Section 2.2) TinySwitch could be used to also enable switchable applications. During the algorithm evaluation step of the TinyCubus workflow (see Section 4.5) the algorithms also have to be tested with all combinations of possible applications.



# Bibliography

- [AMM<sup>+</sup>12] Junaid Ansari, Elena Meshkova, Wasif Masood, Arham Muslim, Janne Riihijärvi, and Petri Mähönen. Confab: Component based optimization of wsn protocol stacks using deployment feedback. In *Proceedings of the 10th ACM International Symposium on Mobility Management and Wireless Access*, MobiWac '12, pages 19–28, New York, NY, USA, 2012. ACM.
- [AS07] Muhammad Abdulla and Robert Simon. Characteristics of common mobility models for opportunistic networks. In *Proc. of the 2nd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, pages 105–109. ACM, Oct 2007.
- [AYO<sup>+</sup>10] Hande Özgür Alemdar, Gökhan Remzi Yavuz, Mustafa Ozan Özen, Yunus Emre Kara, Özlem Durmaz Incel, Lale Akarun, and Cem Ersoy. Multi-modal fall detection within the wecare framework. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 436–437, New York, NY, USA, 2010. ACM.
- [BBB04] J. Burrell, T. Brooke, and R. Beckwith. Vineyard computing: sensor networks in agricultural production. *IEEE Pervasive Computing*, 3(1):38–45, Jan 2004.
- [BHL<sup>+</sup>09] Carlo Alberto Boano, Zhitao He, Yafei Li, Thiemo Voigt, Marco Zuniga, and Andreas Willig. Controllable radio interference for experimental and testing purposes in wireless sensor networks. In *Proc. of the 4th IEEE Intl. Workshop on Practical Issues In Building Sensor Network Applications*, 2009.
- [BHS03] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 187–200, New York, NY, USA, 2003. ACM.
- [BHSR04] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - A Component System for Pervasive Computing. In *Proc. of the 2nd IEEE Conference on Pervasive Computing and Communications (PERCOM'04)*, page 67, 2004.

## Bibliography

- [BKS01] Stephan Börzsöny, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proc. of the 17th Intl. Conf. on Data Engineering*, pages 421–430, 2001.
- [BM11] T. Benmansour and S. Moussaoui. Gmac: Group mobility adaptive clustering scheme for mobile wireless sensor networks. In *10th International Symposium on Programming and Systems (ISPS 2011)*, pages 67–73, April 2011.
- [BNC02] Jeff Boleng, William Navidi, and Tracy Camp. Metrics to enable adaptive protocols for mobile ad hoc networks. In *Proc. of the Intl. Conf. on Wireless Networks (ICWN '02)*, pages 293–298, 2002.
- [BSAH12] N. Bin Shafi, K. Ali, and H.S. Hassanein. No-reboot and zero-flash over-the-air programming for wireless sensor networks. In *9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON 2012)*, pages 371–379, June 2012.
- [CMP<sup>+</sup>09] M. Ceriotti, L. Mottola, G.P. Picco, A.L. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon. Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. In *Proc. of the 8th International Conference on Information Processing in Sensor Networks (IPSN 2009)*, pages 277–288, Apr 2009.
- [CPMW08] Bor-rong Chen, Geoffrey Peterson, Geoff Mainland, and Matt Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *Proc. of the 4th Intl. Conf. on Distributed Computing in Sensor Systems (DCOSS'08)*, pages 79–98, 2008.
- [CPS09] Andrea E. Clementi, Francesco Pasquale, and Riccardo Silvestri. Manets: High mobility can make up for low transmission power. In *Proc. of the 36th Intl. Colloquium on Automata, Languages and Programming (ICALP '09)*, pages 387–398, Jul 2009.
- [CST11] Andrea E. F. Clementi, Riccardo Silvestri, and Luca Trevisan. Information spreading in dynamic graphs. *CoRR*, abs/1111.0583, 2011.
- [DBTV08] Patrick Denantes, Florence Bénézit, Patrick Thiran, and Martin Vetterli. Which distributed averaging algorithm should i choose for my sensor network? In *Proc. of the 27th Conf. on Computer Communications (INFOCOM'08)*, pages 986–994, 2008.
- [Dey01] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, February 2001.
- [DFEV06] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Runtime dynamic linking for reprogramming wireless sensor networks. In *Proc. of the 4th Intl. Conf. on Embedded networked sensor systems*, 2006.

- [DGM05] Amol Deshpande, Carlos Guestrin, and Samuel R. Madden. Resource-aware wireless sensor-actuator networks. *IEEE Data Engineering*, 28(1):40–47, Mar 2005.
- [DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the 29th IEEE Intl. Conf. on Local Computer Networks*, 2004.
- [DLW<sup>+</sup>10] Wei Dong, Yunhao Liu, Xiaofan Wu, Lin Gu, and Chun Chen. Elon: enabling efficient and long-term reprogramming for wireless sensor networks. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and modeling of computer systems*, 2010.
- [DNF<sup>+</sup>05] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D.C. Sicker, and D. Grunwald. MultiMAC - an adaptive MAC framework for dynamic radio networking. In *1st IEEE Intl. Symp. on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, pages 548–555, 2005.
- [DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.
- [DZL<sup>+</sup>10] Hanlin Deng, Baoxian Zhang, Cheng Li, Kui Huang, and Haitao Liu. Max-min aggregation in wireless sensor networks: mechanism and modeling. *Wireless Communications and Mobile Computing*, 2010.
- [EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.
- [FDH<sup>+</sup>09] M. Flatscher, M. Dielacher, T. Herndl, T. Lentsch, R. Matischek, J. Prainsack, W. Pribyl, H. Theuss, and W. Weber. A robust wireless sensor node for in-tire-pressure monitoring. In *IEEE International Solid-State Circuits Conference - Digest of Technical Papers (ISSCC 2009)*, pages 286–287, 287a, Feb 2009.
- [FET<sup>+</sup>10] N. Finne, J. Eriksson, N. Tsiftes, A. Dunkels, and T. Voigt. Improving sensornet performance by separating system configuration from system logic. In *Proc. of the 7th Europ. Conf. on Wireless Sensor Networks*, 2010.
- [FGJ<sup>+</sup>07] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo. TEP 123: The Collection Tree Protocol (CTP). online resource <http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html>, 2007.

## Bibliography

- [FNL09] Carlos M. S. Figueiredo, Eduardo F. Nakamura, and Antonio A. F. Loureiro. A hybrid adaptive routing algorithm for event-driven wireless sensor networks. *Sensors*, 9(9):7287–7307, 2009.
- [FZTS11] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with glossy. In *10th International Conference on Information Processing in Sensor Networks (IPSN 2011)*, pages 73–84, April 2011.
- [GKE04] Benjamin Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (SNACK). In *Proceedings of ACM SenSys '04*, pages 69–80, 2004.
- [GLv<sup>+</sup>03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation*, pages 1–11, 2003.
- [GMN09] Matthias Gauger, Pedro J. Marrón, and Christoph Niedermeier. Tiny-Modules: Code module exchange in TinyOS. In *6th Intl. Conf. on Networked Sensing Systems*, 2009.
- [HC04] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd Intl. Conf. on Embedded Networked Sensor Systems*, 2004.
- [HCB00] W.R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. of the 33rd Annual Hawaii Intl. Conf. on System Sciences (HICCS '00)*, Jan 2000.
- [HHKK04] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, June 2004.
- [HKB99] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 174–185, 1999.
- [HKWW06] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor network. In *Proc. of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality (RealMAN'06)*, 2006.
- [HSW<sup>+</sup>00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.



- [Hun86] J. Stuart Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18(4):203–210, Oct 1986.
- [JVLT<sup>+</sup>09] Xiaofan Jiang, Minh Van Ly, Jay Taneja, Prabal Dutta, and David Culler. Experiences with a high-fidelity wireless building energy auditing network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 113–126, New York, NY, USA, 2009. ACM.
- [KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [KHCL07] Kevin Klues, Gregory Hackmann, Octav Chipara, and Chenyang Lu. A component-based architecture for power-efficient media access control in wireless sensor networks. In *Proc. of the 5th Intl. Conf. on Embedded networked sensor systems*, pages 59–72, 2007.
- [KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for “smart dust”. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, pages 271–278, New York, NY, USA, 1999. ACM.
- [KLC<sup>+</sup>10] Jeonggil Ko, Jong Hyun Lim, Yin Chen, Rvǎzvan Musvaloiu-E, Andreas Terzis, Gerald M. Masson, Tia Gao, Walt Destler, Leo Selavo, and Richard P. Dutton. Medisn: Medical emergency detection in sensor networks. *ACM Trans. Embed. Comput. Syst.*, 10(1):11:1–11:29, Aug 2010.
- [KMO08] R. Krishna, K. McCusker, and N.E. O'Connor. Optimising resource allocation for background modeling using algorithm switching. In *2nd ACM/IEEE Intl. Conf. on Distributed Smart Cameras (ICDCS)*, pages 1–7, 2008.
- [KMR05] Abdelmajid Khelil, Pedro José Marrón, and Kurt Rothermel. Contact-based mobility metrics for delay-tolerant ad hoc networking. In *Proc. of the 13th Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '05)*, pages 435–444, 2005.
- [KNE<sup>+</sup>04] Sachin Kogekar, Sandeep Neema, Brandon Eames, Xenofon Koutsoukos, Akos Ledeczi, and Miklos Maroti. Constraint-guided dynamic reconfiguration in sensor networks. In *Proceedings of the third international symposium on Information processing in sensor networks (IPSN'04)*, pages 379–387, New York, NY, USA, 2004. ACM Press.
- [KVPJ08] G.S. Kumar, M.V. Vinu Paul, and K.P. Jacob. Mobility metric based leach-mobile protocol. In *16th International Conference on Advanced Computing and Communications (ADCOM 2008)*, pages 248–253, Dec 2008.

## Bibliography

- [LAAZ14] A. Laya, L. Alonso, and J. Alonso-Zarate. Is the random access channel of lte and lte-a suitable for m2m communications? a survey of alternatives. *IEEE Communications Surveys Tutorials*, 16(1):4–16, Feb 2014.
- [LC02] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 2002.
- [LM03] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 107–118, 2003.
- [LMM<sup>+</sup>06] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, Matthias Gauger, Olga Saukh, and Kurt Rothermel. TinyXXL: Language and run-time support for cross-layer interactions. In *Proc. of the 3rd IEEE Conf. on Sensor, Mesh and Ad Hoc Communications and Networks*, 2006.
- [LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the 1st Symp. on Networked Systems Design and Implementation*, 2004.
- [LSKT13] Junsong Liao, BrajendraK Singh, MohammedAS Khalid, and KemalE Tepe. FPGA based wireless sensor node with customizable event-driven architecture. *EURASIP Journal on Embedded Systems*, 2013(1), 2013.
- [MALW10] W. Munawar, M.H. Alizai, O. Landsiedel, and K. Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. In *IEEE International Conference on Communications (ICC 2010)*, pages 1–6, May 2010.
- [MCM15] Daniel Minder, Jesús Capitán, and Pedro José Marrón. Smart-action iot international roadmapping. Technical report, University of Duisburg Essen, 2015.
- [MCP<sup>+</sup>02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM.
- [MGL<sup>+</sup>06] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proc. of the 3rd Europ. Workshop on Wireless Sensor Networks*, 2006.

- [MHM10] Daniel Minder, Marcus Handte, and Pedro José Marrón. Tinyadapt: An adaptation framework for sensor networks. In *Proc. of the 7th Internat. Conf. on Networked Sensing Systems (INSS'10)*, pages 253–256, 2010.
- [ML08] David Moss and Philip Levis. BoX-MACs: Exploiting physical and link layer boundaries in low-power networking. Technical Report SING-08-00, Stanford University, 2008.
- [MLM<sup>+</sup>05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Roethermel. Tinycubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 278—289, 2005.
- [MNLL06] Xiaoqiao Meng, Thyaga Nandagopal, Li Li, and Songwu Lu. Contour maps: monitoring and diagnosis in sensor networks. *Comput. Netw.*, 50(15):2820–2838, Oct 2006.
- [MPS08] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. Figaro: Fine-grained software reconfiguration for wireless sensor networks. In *Proc. of the 5th European Conference on Wireless Sensor Networks*, volume 4913 of *Lecture Notes in Computer Science*, pages 286–304. Springer, 2008.
- [MRAM09] Elena Meshkova, Janne Riihijärvi, Andreas Achtzehn, and Petri Mähönen. Exploring simulated annealing and graphical models for optimization in cognitive wireless networks. In *Proceedings of the 28th IEEE Conference on Global Telecommunications, GLOBECOM'09*, pages 4939–4946, November 2009.
- [PJ04] Huan Pham and Sanjay Jha. An adaptive mobility-aware mac protocol for sensor networks (ms-mac). In *IEEE Intl. Conf. on Mobile Ad-hoc and Sensor Systems*, pages 558–560, Oct 2004.
- [PLA] PLANET web site. <http://www.planet-ict.eu/>.
- [PSC05] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Fourth International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 364–369, April 2005.
- [PSJ04] Santashil PalChaudhuri, Amit Kumar Saha, and David B. Johnson. Adaptive clock synchronization in sensor networks. In *Proc. of the 3rd international symposium on Information processing in sensor networks (IPSN '04)*, pages 340–348, Apr 2004.
- [QK06] Liang Qin and Thomas Kunz. Mobility metrics to enable adaptive routing in manet. In *Proc. of the Intl. Conf. on Wireless and Mobile Computing, Networking and Communications (WiMob'06)*, pages 1–8, 2006.

## Bibliography

- [RCK<sup>+</sup>05] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proc. of the 3rd Intl. Conf. on Embedded networked sensor systems (SenSys'05)*, pages 255–267, 2005.
- [RV04] Ruud Riem-Vis. Cold chain management using an ultra low power wireless sensor network. In *Proceedings of the MobiSys Workshop on Applications of Mobile Embedded Systems (WAMES 2004)*, pages 21–23, Boston, MA, USA, Jun 2004.
- [San14] San Francisco Municipal Transportation Agency. Sfpark: Putting theory into practice - pilot project summary and lessons learned. Technical report, Jun 2014.
- [SAZM10] Obaid Salikeen, Junaid Ansari, Xi Zhang, and Petri Mähönen. Enabling flexible mac protocol design for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10*, pages 353–354, New York, NY, USA, 2010. ACM.
- [SGC13] Marcin Szczodrak, Omprakash Gnawali, and Luca P. Carloni. Dynamic reconfiguration of wireless sensor networks to support heterogeneous applications. In *IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2013)*, pages 52–61, May 2013.
- [SMSA14] Tran The Son, Hoa Le Minh, Graham Sexton, and Nauman Aslam. A novel encounter-based metric for mobile ad-hoc networks routing. *Ad Hoc Networks*, 14(0):2 – 14, 2014.
- [SNMT07] Ivan Stoianov, Lama Nachman, Sam Madden, and Timur Tokmouline. Pipenet: a wireless sensor network for pipeline monitoring. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN '07*, pages 264–273, New York, NY, USA, 2007. ACM.
- [SSFM11] Robert Sauter, Olga Saukh, Oliver Frietsch, and Pedro José Marrón. TinyLTS: Efficient network-wide Logging and Tracing System for TinyOS. In *Proc. of the 30th Intl. Conf. on Computer Communications (INFOCOM)*, pages 2033–2041, April 2011.
- [Ste11] Richard A. Stein. Super-spreaders in infectious diseases. *Intl. Journal of Infectious Diseases*, 15(8):e510–e513, 2011.
- [TLP05] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Fourth International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 477–482, Apr 2005.
- [TPS<sup>+</sup>05] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna,

- David Gay, and Wei Hong. A macroscope in the redwoods. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 51–63, New York, NY, USA, 2005. ACM.
- [VFG<sup>+</sup>14] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Harald Sundmaeker, Markus Eisenhauer, Klaus Moessner, Marilyn Arndt, Maurizio Spirito, Paolo Medagliani, Raffaele Giaffreda, Sergio Gusmeroli, Latif Ladid, Martin Serrano, Manfred Hauswirth, and Gianmarco Baldini. Internet of Things Strategic Research and Innovation Agenda. In Ovidiu Vermesan and Peter Friess, editors, *Internet of Things – From Research and Innovation to Market Deployment*, chapter 3, pages 7–142. River Publishers, 2014.
- [WTV<sup>+</sup>07] Georg Wittenburg, Kirsten Terfloth, Freddy López Villafuerte, Tomasz Naumowicz, Hartmut Ritter, and Jochen Schiller. Fence monitoring - experimental evaluation of a use case for wireless sensor networks. In Koen Langendoen and Thiemo Voigt, editors, *Proc. of the 4th European Conference on Wireless Sensor Networks (EWSN 2007)*, volume 4373 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin Heidelberg, Jan 2007.
- [XBJ07] Sanlin Xu, Kim L. Blackmore, and Haley M. Jones. An analysis framework for mobility metrics in mobile ad hoc networks. *EURASIP Journal on Wireless Communication Networks*, 2007(1), Jan 2007.
- [ZFM<sup>+</sup>12] Marco Zimmerling, Federico Ferrari, Luca Mottola, Thiemo Voigt, and Lothar Thiele. ptunes: Runtime parameter adaptation for low-power mac protocols. In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks*, IPSN '12, pages 173–184, New York, NY, USA, 2012. ACM.
- [ZGE02] Yonggang Jerry Zhao, Ramesh Govindan, and Deborah Estrin. Residual energy scan for monitoring sensor networks. In *Proc. of the IEEE Wireless Communications and Networking Conference (WCNC2002)*, pages 356–362, March 2002.
- [ZGE03] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *Proc. of the 1st Intl. Workshop on Sensor Network Protocols and Applications*, pages 139–148, 2003.
- [ZR03] M. Zorzi and R.R. Rao. Geographic random forwarding (geraf) for ad hoc and sensor networks: multihop performance. *IEEE Transactions on Mobile Computing*, 2(4):337–348, Oct 2003.